# Traceability Management for Impact Analysis

Andrea De Lucia, Fausto Fasano, and Rocco Oliveto
*Dipartimento di Matematica e Informatica, University of Salerno*
*Via Ponte don Melillo, 84084 Fisciano (SA), Italy*
{*adelucia, ffasano, roliveto*}*@unisa.it*

## Abstract

*Software change impact analysis is the activity of the software maintenance process that determines possible effects of proposed software changes. This activity is necessary to be aware of ripple-effects caused by the change and record them so that nothing is overlooked. A change has not only impact on the source code, but also on the other related software artefacts, such as requirements, design, and test. For this reason, impact analysis can be efficiently supported through traceability information. In this paper we review traceability management in the context of impact analysis and discuss the main challenges and research directions.*

## 1 Introduction

Software maintenance is defined as "*the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment*" [63]. This is an inevitable process, as a software system has to undergo continual changes to keep its usefulness during its life-cycle [71]. Maintenance activities can be classified as adaptive, perfective, and corrective [64] and each of them involves changes that potentially degrade the software unless it is proactively controlled [71].

The IEEE Standard for Software Maintenance [64] also describes a process composed of several phases ranging from the identification, classification, and prioritization of a modification request to delivery of the modified software system. In particular, after the design and the implementation of the required change, it is necessary to perform regression testing to detect possible faults introduced by the modification [64]. So, in order to analyse the impact of a software change it is needed to ascertain parts of the system that will be affected by the change and examine them for possible further impacts [7, 8, 14]. This activity is necessary to be aware of ripple-effects caused by the change and record them so that nothing is overlooked [120].

While Arnold and Bohner [7] pointed out that there is no definition of software change impact analyis in the IEEE Standard Glossary of Software Engineering Terminology [63], the need for impact analysis is clearly expressed in the software maintenance process defined in the IEEE Standard for Software Maintenance [64], since its first version dated 1992. In particular, software change impact analysis should [64]:

- identify potential ripple effects;
- allow trade-offs between suggested software change approaches to be considered;
- be performed with the help of documentation abstracted from the source code;
- consider the history of prior changes, both successful and unsuccessful.

In particular, a change has not only impact on the source code, but also on the other related software artefacts, such as requirement, design, and test artefacts [8]. For this reason, impact analysis can be efficiently supported through traceability information. Traceability has been defined as "*the ability to describe and follow the life of an artefact, in both a forwards and backwards direction*" [53]. Thus, traceability links help software engineers to understand the relationships and dependencies among various software artefacts. This means that once a maintainer has identified the high-level document (e.g., requirement, use case) related to the feature to be changed, traceability helps to locate the pieces of design, code and whatever need to be maintained.

In this paper we discuss traceability management in the context of impact analysis and survey several approaches proposed in the literature. The paper is organised as follows. Section 2 describes the impact analysis process, while Section 3 classifies different traceability management approaches in the context of impact analysis. Section 4 highlights open issues giving an overview on the state of the art on two of the main challenges in traceability, i.e., link recovery and evolution. Finally, Section 5 gives concluding remarks.

978-1-4244-2655-3/08/$25.00 © 2008 IEEE

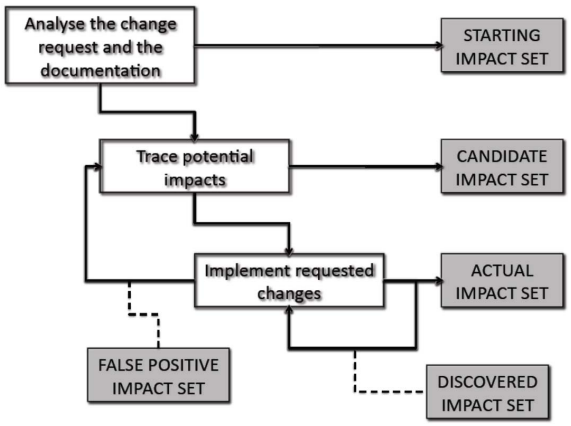


**Figure 1. Impact analysis process [14]**

## 2 Impact Analysis

Figure 1 shows the impact analysis process [7] [14]. The process starts analysing both the change request specifications and the system code and documentation to identify the initial set of software artefacts (e.g., source code, requirements, use cases, test cases, etc) that could be affected by the required change. This set is called *Starting Impact Set* (SIS) [14]. Artefacts in the SIS are then analysed to identify other artefacts estimated to be affected by the change requests. The resulting set of artefacts is the *Candidate Impact Set* (CIS) [14] (also called *Estimated Impact Set* in [7]). Once the change is implemented, the *Actual Impact Set* (AIS) is the set of artefacts actually modified [14]. Note that the AIS is not unique with respect to a change request, since a change can be implemented in several ways.

As show in Figure 1, impact analysis is an iterative process. In particular, during the implementation of a change new impacted artefacts not included in the CIS can be discovered. The set of such artefacts is called the *Discovered Impact Set* (DIS) [14] and represents an under-estimate of impacts. On the other hand, it is also possible that some artefacts in the CIS are not impacted by the change being implemented. The set of these artefacts is called the *False-Positive Impact Set* (FPIS) [14] and represents the over-estimate of impacts in the analysis. The union of CIS and DIS minus the artefacts in FPIS should result in the AIS.

The goal of the impact analysis process is to estimate a CIS that is as close as possible to the AIS. Several metrics can be defined to evaluate the accuracy of the impact analysis process [7, 11, 14, 46, 88]. For example, *Recall* measures the percentage of actual impacts included in CIS (i.e., the ratio between $\mid CIS \cap AIS \mid$ and $\mid AIS \mid$), while *Precision* measures the percentage of candidate impacts that are actual impacts (i.e., the ratio between $\mid CIS \cap AIS \mid$ and $\mid CIS \mid$). Recall is 1 when DIS is empty, while precision is 1 when FPIS is empty, so if CIS coincides with AIS both recall and precision are 1. Unfortunately, this happens very rarely and a trade-off between the two metrics has to be found. Indeed, the larger the CIS the higher the probability to identify all actual impacts, but also the higher the probability that many false positives are identified.

### 2.1 Identifying the Starting Impact Set

The first step of the impact analysis process concerns the identification of the SIS that requires the analysis of the change request specification and both the source code and the software documentation. It is worth noting that the larger the system the higher the effort required to identify the SIS. In particular, it is difficult to map the concepts defined in a change request specification onto source code components. Biggerstaff *et al.* [12] defined the *"concept assignment problem"* as the problem of "*discovering human-oriented concepts and assigning them to their realizations*". Different approaches for the identification of concepts or features in code have been proposed [114]. Some methods are based on static analysis [96], dynamic analysis [1, 115, 117], or a combination of them [45]. However, Biggerstaff *et al.* [12] observed that the concept assignment process cannot be completely automated since concepts and programs are not at the same level of abstraction. Thus, the human interaction is necessary. They also present an approach where static analysis and both formal and informal information (for example names in the program) are used to locate concepts in the program [12]. The analysis of informal information plays an important role in the identification of concepts in the source code [22]. As revealed by several studies on users' behaviour, such information has been used for years by software engineers dealing with unfamiliar software [104]. The informal information extracted from the source code can be formalised in a Concern Graph that abstracts the implementation details of a concern making explicit the relationships between different parts of the concern [100].

The analysis of informal information has also been addressed using Information Retrieval (IR) techniques [9, 39]. These approaches allow users to run natural language queries (representing a change request) in order to find possibly affected components in the source code [56, 79, 80, 94, 92, 105], in the documentation [2], in CVS comments and change logs [18, 23]. IR techniques can also be combined with static and dynamic analysis in order to improve the retrieval performances of the basic IR models. In particular, Latent Semantic Indexing (LSI) is combined with Formal Concept Analysis (FCA) [93] and with scenario-based probabilistic ranking (SPR) [91], while a light-weight Natural Language Processing (NLP) technique is combined with source code static analysis in [103].

### 2.2 Identifying the Candidate Impact Set

The second step of the impact analysis process is the identification of the candidate impact set: the starting impact set is enhanced with software artefacts that might be affected due to changes made to artefacts in the SIS. Indeed, the need for impact analysis during software maintenance derives from the fact that even a small change in a software system may affect many other parts of the system, thus causing a ripple-effect [119, 120]. As highlighted by Bohner [14], ripple-effects may cause direct or indirect impacts on software artefacts. A direct impact occurs when the artefact is affected due to one of the dependencies that fan-in/out directly to/from the changing artefact, while an indirect impact occurs when the artefact is affected due a set of dependencies forming an acyclic path between the subject artefact and the changing artefact. Thus, starting from the SIS, the direct impacts can be obtained from the dependency graph (represented as a traceability matrix), while the indirect impacts can be achieved by computing the reachability graph. The resulting graph can be visualised or even interactively browsed [34, 61, 68, 81, 107, 116] during an incremental definition of the CIS [46, 96, 122].

It is worth noting that other parts of the system that are affected and are not reachable from the identified starting impact set are not obvious or easy to detect [14], thus an under-estimate of the starting impact set might reduces the recall of the impact analysis process. Moreover, the reachability graph generally indicates potential impacts to an artefact. Thus, there is the risk to over-estimate the candidate impact set if reachability analysis is performed without using any pruning strategy, thus resulting in a low precision. A simple mechanism to reduce the number of false positive is to consider the distance between artefacts. In this way, only artefacts with a distance from the modified artafect less than or equal to a fixed threshold are considered [14]. Associating traceability links with specific semantics [14, 98] or probabilities indicating the likelihood of an indirect impact [108] is also useful to reduce the candidate impact set. Another way to reduce the number of false positives is to define propagation rules and use such rules to navigate the reachability graph in order to define the CIS [10]. A similar approach consists of propagating events from an artefact to the dependent artefacts [24, 29, 34].

Another way to reduce the number of false positives is to incrementally analyse the candidate impact set [46, 96, 122]. In particular, for every artefact in the starting impact set, the software engineer visits all the artefacts interacting with it. Only artefacts actually impacted by the change request are added to the impact set. The software engineer continues this process recursively until he/she identifies all the impacted artefacts [96]. Another way to incrementally define the CIS is based on the idea that the implementation of a change request will modify the set of documents starting from the highest-level document affected by the changes, and propagating the changes down through the full set of documents. Thus, the definition of the CIS can be performed step-by-step starting from the most abstract artefacts of the system model down to the source code artefacts to be changed [46].

**Table 1. Traceability dimensions (adapted from [11])**

| Dimension | Cathegories | |
|---|---|---|
| $D_1$ | Vertical | Horizonthal |
| $D_2$ | Structural | Knowledge based |
| $D_3$ | Implicit | Explicit |

## 3 Traceability

As shown in Section 2, an appropriate support for traceability can determine the effectiveness of the change impact analysis process. In this section we focus on the various aspects that characterise traceability management approaches with respect to their support for impact analysis. We refer to the conceptual framework proposed by Bianchi *et al.* [11] and summarised in Table 1. As discussed in the following, the three dimensions included in this framework concern the type of artefacts involved in the traceability links, the information source used to derive traceability links, and whether these links are explicitely or implicitely stored.

### 3.1 Vertical vs Horizontal Traceability

The first dimension we consider is related to the support for vertical and horizontal traceability. Vertical traceability refers to the abiity to trace dependent artefacts within a model, while horizontal traceability refers to the ability to trace artefacts between different models [72, 88]. Different types of artefacts can be considerd during the change impact analysis process, including requirement, analysis, design, rationale, source code, and testing artefacts.

Most of the approaches that provide vertical traceability information are focused on source code artefacts [3, 17, 52, 98]. In this context, source code analysis techniques [13, 51] are used to detect and capture dependencies among the source code components. As an example, *backward program slicing* [112] uses intraprocedural or interprocedural control and data flow information to identify the statements and predicates that directly or indirectly affect the computation of the values of some program variables at a given program point. On the other hand, *forward slicing* is used to identify the statements and predicates affected by

the computation of the value of some variable at a given program point [62]. Often intraprocedural or interprocedural control and data dependencies are maintained in some form of dependence graph and program slices are computed using graph traversal algorithms [47, 58, 62]. Besides static slicing that is based on static analysis techniques, other forms of slicing including *dynamic slicing* [66], *quasi static slicing* [109], and *conditioned slicing* [20] have been defined that allows to reduce the set of statements included in a program slice.

Reverse Engineering techniques also allow to identify dependencies between modules at the architectural level [21, 26]. Despite the identification of the source code components impacted by a proposed change is generally considered the most important output of the impact analysis process, other approaches address vertical traceability on software artefacts at a higher level of abstraction, for example between documentation elements [108], requirement artefacts [29, 33, 60, 90] or UML models [15, 76].

Vertical traceability provides only a limited view on the artefacts potentially impacted by a proposed change. Complex and large-size software systems require the production of software models with several levels of abstraction (source code can be considered as one of these levels of abstraction) and related artefacts belonging to the different levels need to be changed consistently. As a consequence, many traceability approaches extend vertical traceability with horizontal traceability, thus allowing the management of dependencies between models at different levels of abstraction, e.g. between requirements and design artefacts [31, 35, 75, 102, 110, 124], between requirements and source code [4, 35, 43, 55, 73, 78, 89, 99, 102], between requirements and test cases [35, 75, 89, 102], between design artefacts and source code [4, 5, 57, 78, 82, 87, 101, 111], between requirements or design documents to defect reports [118]. Often traceability between artefacts at different abstraction levels are linked through information elements that are not included in software models but that are needed to document the rationale behind decisions made during the software development and maintenance process [27, 32, 67, 69, 70, 97, 116].

It is important to note that the granularity of the artefacts plays an important role in the definition of traceability links among artefacts. Indeed, the decomposition of artefacts into fine grained sub-artefacts enables a more precise impact analysis and reduces the time needed to identify the exact part of the artefact impacted by the change [16, 24, 86].

### 3.2 Structural vs Knowledge-based Traceability

The second dimension considers the nature of the information source used to derive traceability links. We distinguish between *structural* and *kwnowledge-based* links. Structural links can be usually derived by analysing the artefacts with respect to the syntax and semantics of the formal language in which such artefacts are expressed. For example, use, inheritance, or composition relationships between the classes of an object-oriented program can be easily derived by parsing the source code according to the formal grammar of the object-oriented programming language. Despite they have been mainly applied to source code [17, 52, 98], static analysis techniques can be applied to any formalism with a sufficient formal grammar [95], such as XML documents [44, 54, 85, 113]. As an example, task-specific UML sub-models can be automatically extracted from a class diagram [65] or use cases dependences can be extracted from use case diagrams [48]

On the other hand, knowledge-based traceability refers to artefact dependencies that cannot be automatically derived by parsing the source code or other artefacts developed according to a formal syntax and semantics. Indeed, dealing with a large amount of informal information, structuring and relating this information can only be done by human experts [90]. Different approaches have been proposed to elicit traceability links by exploiting some heuristics rules together with knowledge of the application and/or solution domain. Examples include name tracing rules [3, 5, 82, 110, 111], dynamic analysis rules [43], logic based rules [15, 101], or a combination of syntactic and heuristic rules [124]. Other approaches exploits information retrieval techniques, e.g. [9, 39], to derive traceability links between artefacts with high textual similarity [4, 31, 33, 36, 41, 60, 74, 75, 78, 99, 102, 118, 125], or version history mining techniques to identify logical coupling between artefacts that are changed at the same time [18, 49, 50, 121, 123].

It is worth noting that experts give a confidence indication on the correctness of the link. As a consequence, there might be links that are erroneously identified by using a knowledge-based approach (thus adding artefacts to the False Positive Impact Set) and, conversely, some of the links might be overlooked or missed during the impact analysis process (thus resulting in artefacts in the Discovered Impact Set). As discussed in Section 4.1, this issue is one of the major traceability challenges together with the evolution of traceability links.

### 3.3 Implicit vs Explicit Traceability

The third dimension considered concerns with the representation of the links. In particular, a distinction can be done between the approaches that use *explicit* links to represent a dependence [24, 29, 77, 86, 87, 90, 106] and the approaches that recover them on-the-fly when asked (links are *implicit* in the artefacts) [4, 31, 33, 43, 60, 74, 75, 78, 99, 110, 124].

It is also possible to consider *hybrid* approaches, where some type of links are stored explicitly while other types of links are recovered when needed [17, 35, 55, 89, 97]. Source code analysis and reverse engineering techniques also store program dependencies in some program representation form [19, 40, 58, 62] and traverse it to infer dependencies at a higher abstraction level, that are also stored in the repository [19, 25, 83, 84].

# 4 Traceability Challenges

As discussed in Section 3, one of the main challenges of traceability management is recovering knowledge-based traceability links [30]. Indeed, while techniques to recover structural links are quite mature and links are recovered mostly automatically, techniques to recover knowledge-based links require user validation. The lack of completely automatic approaches also makes the management of the evolution of knowledge-based links more challenging. Explicit traceability information may become soon out of date due to the evolutionary nature of the software. In particular, the traceability matrix could become incomplete (some new correct link is missed) or might include links that are no longer valid. Such a situation is due to the fact that maintaining traceability information up-to-date is a tedious and time consuming task and often is sacrificed under the time pressure of on-going work [53].

## 4.1 Traceability recovery

Several methods have been proposed to recover traceability links between software artefacts of different types. The proposed approaches can be classified according to the method adopted to derive knowledge-based links between artefacts: (i) heuristic-based; (ii) IR-based; and (iii) data mining based.

### 4.1.1 Heuristic-based approaches

Software reflexion model techniques [82] have been used to help a software engineer to compare artefacts by summarising where one artefacts (such as a design) is consistent with and inconsistent with another artefact (such as source). In particular regular expressions are used to exploit naming conventions and map source code model entities onto high-level model entities [5, 15, 82, 101, 111].

More complicated rules can also be derived considering textual documents written in natural language. In particular, it is possible to define rules that syntactically match related terms in the textual parts of the requirement artefacts with related elements in an object model (e.g. classes, attributes, operations). Thus, traceability relations of different types can be defined when a match is found [124]. Recently, the use of ontologies has also been proposed for recovering traceability links [99].

Other heuristics can be derived analysing the guidelines for changing requirements and design documents. Obviously, such rules can be used to recover relationships between requirements and design artefacts. The dynamic analysis is also used to define tracing heuristics. In particular, traceability links between requirements and source code can be retrieved by monitoring the source code to record which program classes are used when scenarios are executed [43].

### 4.1.2 IR-based approaches

Several authors have applied Information Retrieval (IR) methods [9, 39] to the problem of recovering traceability links between software artefacts of different types. The rationale behind such a choice is that most of the documentation that accompany large software systems consists of free text documents expressed in a natural language and high textual similarity between two artefacts might highlight the presence of a traceability link.

Antoniol *et al.* [4] were the first to use such techniques to recover traceability links between source code and and high-level documentation (i.e., requirements and manual pages). They applied the probabilistic and vector space models [9] and the experimental results did not highlight any evidence to prefer one approach over the other. Other authors have used enhancing strategies to improve the tracing accuracy of these basic models. Examples of enhancing strategies for the probabilistic model are hierarchical modelling [31], logical clustering of artefacts [31, 42], semi-automated pruning of the probabilistic network [31], and query term coverage and phrasing [125]. Instead, thesaurus [59, 102], key-phrases [59] and pivot normalisation weighting scores [102] have been proposed to improve the accuracy of the vector space model.

Marcus and Maletic [78] were the first to use Latent Semantic Indexing (LSI) [39] to recover traceability links between source code and documentation. They compared the accuracy of LSI with respect to the vector space and probabilistic models showing how the latter models require morphological analysis of text contained in source code and documentation to achieve the same accuracy as LSI. LSI was also used to reconstruct traceability links between high-level and low-level requirements [60], as well as among software artefacts of different types [35, 75].

The retrieval accuracy of an IR-based traceability recovery tool can be improved by learning from an input training set of correct links [6, 41] or from user feedback provided during the classification of the candidate links [36, 60]. However, even though the retrieval accuracy generally improves with the use of feedback, IR-based approaches are

still far from solving the problem of recovering all correct links with a low classification effort [36].

Despite the IR method used, all these traceability recovery approaches compute the similarity between all possible pairs of artefacts and return a ranked list ordered by decreasing similarity. Generally, this list contains a higher density of correct traceability links in the upper part of the list and a much lower density of such links in the bottom part of the list. As a consequence, while IR-based approaches help in the identification of traceability links in the upper part of the ranked list, the ranking in the lower part of the returned list of candidate links give almost no help, due to the presence of too many links that are not correct (*false positives*). This means that the effort required to discard false positves becomes much higher than the effort to validate correct links. For this reason, most approaches use some method to cut (or filter) the ranked list (e.g., a threshold on the similarity value), thus presenting the software engineer only the subset of top links in the ranked list [4, 35, 60, 74, 75, 78]. Of course, the lower the similarity threshold used, the higher the number of correct links as well as the number of false positives retrieved. De Lucia *et al.* [35] also propose to incrementally decrease the similarity threshold to give the software engineer the control on the number of validated correct links and the number of discarded false positives. In this way, the process can be stopped when the effort to discard false positives is becoming much higher than the effort to identify new correct links.

In the last decade several IR-based tools have also been proposed to support the software engineer during the traceability recovery process [28, 35, 60, 74, 81]. In some cases, the usefulness of such tools has also been assessed through user studies. The analysis of the achieved results revealed that an IR-based tool represents a valuable support during the traceability link identification [35]. Moreover, the tool significantly reduces the time spent by the software engineer to complete the task and the tracing errors [37]. Also, experiments show that the incremental process reduces the effort to classify proposed links with respect to a "one-shot" approach, where the full ranked list of links is proposed without similarity information and filtering [38].

#### 4.1.3 Data mining-based approaches

Data mining techniques on software configuration management repositories have been used to recover traceability links between source code artefacts. Gall *et al.* [49] were the first to use release data to detect logical coupling between modules. They used CVS history to detect fine-grained logical coupling between classes, files, and functions. Their methodology investigated the historical development of classes measuring the time when new classes are added to the system and when existing classes are changed and maintaining attributes related to changes of classes, such as the author or the date of a change. Such information was inspected to reveal common change behaviour of different parts of the system during the evolution (referred to as logical coupling) [50].

Ying *et al.* [121] used association rule mining on CVS version archives. Their approach was based on the mining of change patterns (files that were changed together frequently in the past) from the source code change history of the system. Mined change patterns were used to recommend possible relevant files as a developer performs a modification task. Similarly, Zimmermann *et al.* [123] developed an approach that also uses association rule mining on CVS data to recommend source code that is potentially relevant to a given fragment of source code. The rules determined by their approach can describe change associations between fine-grained program entities, such as functions or variables, as well as coarse-grained entities, such as classes or files. Coarse-grained rules have a higher support count and usually return more results. However, they are less precise in the location and, thus, only of limited use for guiding programmers.

### 4.2 Link evolution

Another important traceability management issue concerns the evolution of traceability links. Nistor *et al.* [87] developed ArchEvol an environment that manages the evolution of architecture-to-implementation traceability links throughout the entire software life cycle. The proposed solution maintains the mapping between architecture and code and ensures that the right versions of the architectural components map onto the right versions of the code (and vice versa), when changes are made either to the architecture or to the code.

Nguyen *et al.* [86] developed Molhado, an architectural configuration management system that automatically updates traceability links between architecture and code artefacts during software evolution. Molhado uses a single versioning mechanism for all software components and for the connections between them and can track changes at a very fine-grained level, allowing users to return to a consistent state of a single node or link. Maletic *et al.* [76] proposed an approach to support the evolution of traceability links between source code and UML artefacts. The authors used an XML-based representation for both the source code and the UML artefacts and applied meta-differencing whenever an artefact is checked-in to identify specific changes and identify traceability links that might have been affected by the change.

Antoniol *et al.* [3] proposed a method based on string edit distance to assess the compliance of OO software systems across different releases. The method has been applied

to two case studies: the first case study has been used to assess the parameters, i.e., the matching weights and the pruning threshold. The second case study has been conducted to further validate the proposed method. The parameters of the method have been calibrated on freely available and public domain software written in C++. Moreover, using the method on different software is likely to require a parameter re-calibration.

Another approach to monitor link evolution is based on the analysis of textual similiarity between traced artefacts [35]. In particular, during software development and evolution there could be links traced between software artefacts with a low similarity. Such a situation might indicate that the link was erroneusly traced or that it is not valid anymore. On the other hand, low similarity between traced artefacts might also be an indication of low artefact quality, in terms of textual description in high level artefacts or misuse of identifiers and poor commenting in source code artefacts.

## 5 Conclusion

The identification of the set of artefacts that would be impacted as a consequence of processing a change request, is commonly based on two sources of information: domain expert knowledge of the system and artefact traceability. The former can be obviously used to build or improve the traceability infrastructure. In this paper, we addressed the support provided by traceability management during the software change impact analysis.

A survey of the approaches proposed to support impact analysis based on the use of traceability information has been presented. Among these, we distinguished among three different dimensions, taking into considerations the type of traceability links (vertical or horizontal), the source of information used to derive them (structural or knowledge based), and their representation within the traceability management system (implicit or explicit).

We also outlined the major challenges in traceability management, namely traceability recovery and link evolution.

## References


[1] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *IEEE Comp.*, 31(7):64–73, 1998.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Identifying the starting impact set of a maintenance request. In *Proc. $4^{th}$ Europ. Conf. on Softw. Maint. and Reeng.*, pages 227–230. IEEE CS Press, 2000.

[3] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Softw. - Pract. and Exp.*, 31(4):331–355, 2001.

[4] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE TSE*, 28(10):970–983, 2002.

[5] G. Antoniol, B. Caprile, A. Potrich, and P. Tonella. Design-code traceability for object-oriented systems. *Annals of Softw. Eng.*, 9(1-4):35–58, 2000.

[6] G. Antoniol, G. Casazza, and A. Cimitile. Traceability recovery by modelling programmer behaviour. In *Proc. $7^{th}$ Working Conf. on Rev. Eng.*, pages 240–247. IEEE CS Press, 2000.

[7] R. S. Arnold and S. A. Bohner. Impact analysis – towards a framework for comparison. In *Proc. $9^{th}$ Conf. on Softw. Maint.*, pages 292–301, 1993.

[8] R. S. Arnold and S. A. Bohner (eds.). *Software Change Impact Analysis*. Wiley-IEEE CS Press, 1996.

[9] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[10] S. Barros, T. Bodhuin, A. Escudie, J. Queille, and J. Voidrot. Supporting impact analysis: a semi-automated technique and associated tool. In *Proc. $11^{th}$ ICSM*, pages 42–51. IEEE CS Press, 1995.

[11] A. Bianchi, A. R. Fasolino, and G. Visaggio. An exploratory case study of the maintenance effectiveness of traceability models. In *Proc. $8^{th}$ Int'nl Workshop on Program Compr.*, pages 149–158. IEEE CS Press, 2000.

[12] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communication of ACM*, 37(5):72–82, 1994.

[13] D. Binkley. Source code analysis: A road map. In *Future of Softw. Eng.*, pages 104 – 119. IEEE CS Press, 2007.

[14] S. A. Bohner. Software change impacts: An evolving perspective. In *Proc. $18^{th}$ ICSM*, pages 263–272, 2002.

[15] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact analysis and change management of UML models. In *Proc. $19^{th}$ ICSM*, pages 256–265. IEEE CS Press, 2003.

[16] B. Brügge, A. De Lucia, F. Fasano, and G. Tortora. Supporting distributed software development with fine-grained artefact management. In *Proc. $1^{st}$ Int'nl Conf. on Global Softw. Eng.*, pages 213–222. IEEE CS Press, 2006.

[17] J. Buckner, J. Buchta, M. Petrenko, and V. Rajlich. JRipples: A tool for program comprehension during incremental change. In *Proc. $13^{th}$ Int'nl Workshop on Program Compr.*, pages 149–152. IEEE CS Press, 2005.

[18] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proc. $11^{th}$ Int'nl Symp. on Softw. Metrics*, pages 20–29. IEEE CS Press, 2005.

[19] G. Canfora, A. Cimitile, U. De Carlini, and A. De Lucia. An extensible system for source code analysis. *IEEE TSE*, 24(9):721–740, 1998.

[20] G. Canfora, A. Cimitile, and A. D. Lucia. Conditioned program slicing. *Inf. and Softw. Tech.*, 40(11-12):595–607, 1998.

[21] G. Canfora and M. Di Penta. New frontiers of reverse engineering. In *Future of Softw. Eng.*, pages 326–341. IEEE CS Press, 2007.

[22] B. Caprile and P. Tonella. Nomen est omen: Analyzing the language of function identifiers. In *Proc. $6^{th}$ Working Conf. on Rev. Eng.*, pages 112–122. IEEE CS Press, 1999.

[23] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: searching through source code using CVS comments. In *Proc. $17^{th}$ ICSM*, pages 364–373. IEEE CS Press, 2001.

[24] J. Y. J. Chen and S. C. Chou. Consistency management in a process environment. *J. of Syst. and Softw.*, 47(2-3):105–110, 1999.

[25] Y. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE TSE*, 16(3):325–334, 1990.

[26] E. Chikofsky and J. I. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[27] A. Cimitile, F. Lanubile, and G. Visaggio. Traceability based on design decisions. In *Proc. $8^{th}$ Conf. on Softw. Maint.*, pages 309–317. IEEE CS Press, 1992.

[28] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *IEEE Comp.*, 40(6):27–35, 2007.

[29] J. Cleland-Huang, C. K. Chang, and M. J. Christensen. Event-based traceability for managing evolutionary change. *IEEE TSE*, 29(9):796–810, 2003.

[30] J. Cleland-Huang, A. Dekhtyar, J. Hayes, G. Antoniol, B. Berenbach, A. Eyged, S. Ferguson, J. Maletic, and A. Zisman. Grand challenges in traceability. Technical Report COET-GCT-06-01-0.9, Center of Excellence for Traceability, September 2006.

[31] J. Cleland-Huang, R. Settimi, C. Duan, and X. Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Proc. $13^{th}$ Int'nl Req. Eng. Conf.*, pages 135–144. IEEE CS Press, 2005.

[32] J. Conklin and M. L. Begeman. gIBIS: a hypertext tool for exploratory policy discussion. *ACM TOIS*, 6(4):303–331, 1988.

[33] J. Dag, B. Regnell, P. Carlshamre, M. Andersson, and J. Karlsson. A feasibility study of automated natural language requirements analysis in market-driven development. *Req. Eng.*, 7(1):20–33, 2002.

[34] A. De Lucia, F. Fasano, R. Francese, and R. Oliveto. Traceability management in ADAMS. In *Proc. $1^{st}$ Int'nl Workshop on Distr. Softw. Develop.*, pages 135–149, 2005.

[35] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software aftefact management systems using information retrieval methods. *ACM TOSEM*, 16(4), 2007.

[36] A. De Lucia, R. Oliveto, and P. Sgueglia. Incremental approach and user feedbacks: a silver bullet for traceability recovery. In *Proc. $22^{nd}$ ICSM*, pages 299–309. IEEE CS Press, 2006.

[37] A. De Lucia, R. Oliveto, and G. Tortora. Recovering traceability links using information retrieval tools: a controlled experiment. In *Proc. Int'nl Symp. on Grand Challenges in Traceability*, pages 46–55. ACM Press, 2007.

[38] A. De Lucia, R. Oliveto, and G. Tortora. Assessing IR-based traceability recovery tools through controlled experiments. In *Proc. $23^{rd}$ Int'nl ASE Conf.* IEEE CS Press, 2008.

[39] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *J. of the American Society for Inf. Science*, 41(6):391–407, 1990.

[40] P. T. Devanbu. GENOA - a customizable, front-end-retargetable source code analysis framework. *ACM TOSEM*, 8(2):177–212, 1999.

[41] M. Di Penta, S. Gradara, and G. Antoniol. Traceability recovery in RAD software systems. In *Proc. $10^{th}$ Int'nl Workshop in Program Compr.*, pages 207–216. IEEE CS Press, 2002.

[42] C. Duan and J. Cleland-Huang. Clustering support for automated tracing. In *Proc. $22^{nd}$ Int'nl ASE Conf.*, pages 244–253. ACM Press, 2007.

[43] A. Egyed and P.Grunbacher. Automating requirements traceability: Beyond the record and replay paradigm. In *Proc. $17^{th}$ Int'nl ASE Conf.*, pages 163–171. IEEE CS Press, 2002.

[44] M. Eichberg, M. Mezini, K. Ostermann, and T. Schafer. Xirc: A kernel for cross-artifact information engineering in software development environments. In *Proc. $11^{th}$ Working Conf. on Rev. Eng.*, pages 182–191. IEEE CS Press, 2004.

[45] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE TSE*, 29(3):210–224, 2003.

[46] A. R. Fasolino and G. Visaggio. Improving software comprehension through an automated dependency tracer. In *Proc. $7^{th}$ Int'nl Workshop on Program Compr.*, pages 58–65. IEEE CS Press, 1999.

[47] J. Ferrante, K. Ottenstain, and J. Warren. The program dependence graph and its use in optimization. *ACM TOPLAS*, 9(3):319–349, 1987.

[48] O. Funkhouser, L. H. Etzkorn, and J. William E. Hughes. A lightweight approach to software validation by comparing UML use cases with internal program documentation selected via call graphs. *Softw. Quality Control*, 16(1):131–156, 2008.

[49] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. $14^{th}$ ICSM*, pages 190–198. IEEE CS Press, 1998.

[50] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proc. $6^{th}$ Int'nl Workshop on Princ. of Softw. Evol.*, pages 13–23. IEEE CS Press, 2003.

[51] K. Gallagher and D. Binkley. Program slicing. In *Frontiers of Softw. Maint.* IEEE CS Press, 2008.

[52] K. Gallagher and J. Lyle. Using program slicing in software maintenance. *IEEE TSE*, 17(8):751–761, 1991.

[53] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. $1^{st}$ Int'nl Conf. on Req. Eng.*, pages 94–101, 1994.

[54] T. Grabs and H.-J. Schek. Flexible information retrieval on XML documents. In *LNCS - Intelligent Search on XML Data*, pages 95–106. Springer Berlin, 2003.

[55] M. Grechanik, K. S. McKinley, and D. E. Perry. Recovering and using use-case-diagram-to-source-code traceability links. In *Proc. $6^{th}$ ESEC/FSE Conf.*, pages 95–104. ACM Press, 2007.

[56] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. 23rd ICSE*, pages 265–274. IEEE CS Press, 2001.

[57] J. Han. TRAM: a tool for requirements and architecture management. In *Proc. 24th Australasian Conf. on Comp. science*, pages 60–68. IEEE CS Press, 2001.

[58] M. Harrold and B. Malloy. A unified interprocedural program representation for a maintenance environment. *IEEE TSE*, 19(6):584–593, 1993.

[59] J. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Proc. 11th Int'nl Req. Eng. Conf.*, pages 138–147. IEEE CS Press, 2003.

[60] J. H. Hayes, A. Dekhtyar, and S. K. Sundaram. Advancing candidate link generation for requirements tracing: The study of methods. *IEEE TSE*, 32(1):4–19, 2006.

[61] T. Hildenbrand, M. Geisser, L. Klimpke, and T. Acker. Designing and implementing a tool for distributed collaborative traceability and rationale management. In *Proc. PRIMIUM Subconference at the Multikonferenz Wirtschaftsinformatik*, volume 328. CEUR-WS, 2008.

[62] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[63] IEEE – The Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology*, 1990.

[64] IEEE – The Institute of Electrical and Electronics Engineers, Inc. *IEEE Std 1219-1998: IEEE Standard for Software Maintenance*, 1998.

[65] H. Kagdi, J. I. Maletic, and A. Sutton. Context-free slicing of UML class models. In *Proc. 21st ICSM*, pages 635–638. IEEE CS Press, 2005.

[66] B. Korel and J. Laski. Dynamic slicing of computer programs. *J. of Syst. and Softw.*, 13(3):187–195, 1990.

[67] F. Lanubile and G. Visaggio. Decision-driven maintenance. *J. of Softw. Maint. - Res. and Pract.*, 7(2):91–115, 1995.

[68] M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE TSE*, 29(9):782–795, 2003.

[69] M. Lease, M. Lively, and J. Leggett. Using an issue-based hypertext system to capture the software life-cycle process. *Hypermedia*, 2(1):29–46, 1990.

[70] J. Lee. SIBYL: a tool for managing group design rationale. In *Proc. ACM Conf. on CSCW*, pages 79–92. ACM Press, 1990.

[71] M. M. Lehman and L. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[72] M. Lindvall and K. Sandahl. Practical implications of traceability. *Softw. - Pract. and Exp.*, 26(10):1161–1180, 1996.

[73] M. Lindvall and K. Sandahl. Traceability aspects of impact analysis in object-oriented systems. *J. of Softw. Maint. - Res. and Pract.*, 10(1):37–57, 1998.

[74] M. Lormans, H. Gross, A. van Deursen, R. van Solingen, and A. Stehouwer. Monitoring requirements coverage using reconstructed views: An industrial case study. In *Proc. 13th Working Conf. on Rev. Eng.*, pages 275–284. IEEE CS Press, 2006.

[75] M. Lormans and A. van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Proc. 10th Europ. Conf. on Softw. Maint. and Reeng.*, pages 45–54. IEEE CS Press, 2006.

[76] I. J. Maletic, M. L. Collard, and B. Simoes. An XML based approach to support the evolution of model-to-model traceability links. In *Proc. 3rd ACM Int'nl Workshop on Traceability in Emerging Forms of Softw. Eng.*, pages 67–72, 2005.

[77] J. I. Maletic, E. Munson, A. Marcus, and T. Nguyen. Combining traceability link recovery with conformance analysis via a formal hypertext model. In *Proc. 2nd Int'nl Workshop on Traceability in Emerging Forms of Softw. Eng.*, pages 47–54. IEEE CS Press, 2003.

[78] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. 25th ICSE*, pages 125–135. IEEE CS Press, 2003.

[79] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proc. 16th Int'nl ASE Conf.*, pages 107–114. IEEE CS Press, 2001.

[80] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proc. 11th Working Conf. on Rev. Eng.*, pages 214–223. IEEE CS Press, 2004.

[81] A. Marcus, X. Xie, and D. Poshyvanyk. When and how to visualize traceability links? In *Proc. 3rd Int'nl Workshop on Traceability in Emerging Forms of Softw. Eng.*, pages 56–61. ACM Press, 2005.

[82] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE TSE*, 27(4):364–380, 2001.

[83] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos: Representing knowledge about information systems. *ACM TOIS*, 8(4):325–362, 1990.

[84] H. A. Mller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *J. of Softw. Maint. - Res. and Pract.*, 5(4):181–204, 1993.

[85] H.-S. Na, O.-H. Choi, and J.-E. Lim. A metamodel-based approach for extracting ontological semantics from UML models. In *LNCS*, volume 4255, pages 411–422. Springer Berlin, 2006.

[86] T. N. Nguyen, E. V. Munson, J. T. Boyland, and C. Thao. Infrastructures for development of object-oriented configuration management services. In *Proc. 27th ICSE*, pages 215–224. ACM Press, 2005.

[87] E. C. Nistor, J. R. Erenkrantza, S. A. Hendrickson, and A. van der Hoek. ArchEvol: Versioning architectural-implementation relationships. In *Proc. 12th Int'nl Workshop on Softw. Config. Mgmt*, pages 99–111. ACM Press, 2005.

[88] S. L. Pfleeger and S. A. Bohner. A framework for software maintenance metrics. In *Proc. 6th Conf. on Softw. Maint.*, pages 320–327, 1990.

[89] F. Pinheiro and J. Goguen. An object-oriented tool for tracing requirements. *IEEE Softw.*, 13(2):52–64, 1996.

[90] K. Pohl. PRO-ART: Enabling requirements pre-traceability. In *Proc. 2nd Int'nl Conf. on Req. Eng.*, pages 76–84. IEEE CS Press, 1996.

[91] D. Poshyvanyk, Y. Gael-Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE TSE*, 33(6):420–432, 2007.

[92] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS - an eclipse plug-in for source code exploration. In *Proc. 14$^{th}$ Int'nl Conf. on Program Compr.*, pages 252–255. IEEE CS Press, 2006.

[93] D. Poshyvanyk and D. Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. 15$^{th}$ Int'nl Conf. on Program Compr.*, pages 37–48. IEEE CS Press, 2007.

[94] D. Poshyvanyk, M. Petrenko, A. Marcus, X. Xie, and D. Liu. Source code exploration with Google. In *Proc. 22$^{nd}$ ICSM*, pages 334–338. IEEE CS Press, 2006.

[95] J.-P. Queille, J.-F. Voidrot, N. Wilde, and M. Munro. The impact analysis task in software maintenance: A model and a case study. In *Proc. 10$^{th}$ ICSM*, pages 234–242. IEEE CS Press, 1994.

[96] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Softw.*, 21(4):62–69, 2004.

[97] B. Ramesh and V. Dhar. Supporting systems development using knowledge captured during requirements engineering. *IEEE TSE*, 9(2):498–510, 1992.

[98] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *ACM SIGPLAN Notices*, 39(10):432–448, 2004.

[99] J. Rilling, R. Witte, and Y. Zhang. Automatic traceability recovery: An ontological approach. In *Proc. Int'nl Symp. on Grand Challenges in Traceability*, pages 66–75. ACM Press, 2007.

[100] M. Robillard and G. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proc. 24$^{th}$ ICSE*, pages 406–416. ACM Press, 2002.

[101] M. Sefika, A. Sane, and R. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proc. 16$^{th}$ ICSE*, pages 387–396. IEEE CS Press, 1996.

[102] R. Settimi, J. Cleland-Huang, O. B. Khadra, J. Mody, W. Lukasik, and C. DePalma. Supportinsoftw.re evolution through dynamically retrieving traces to UML artifacts. In *Proc. 7$^{th}$ Int'nl Workshop on Princ. of Softw. Evol.*, pages 49–54. IEEE CS Press, 2004.

[103] D. Shepherd, Z. Fry, E. Gibson, L. Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proc. 6$^{th}$ Int'nl Conf. on Aspect Oriented Softw. Develop.*, pages 212–224. ACM Press, 2007.

[104] S. Sim, C. Clarke, and R. Holt. Archetypal source code searches: a survey of software developers and maintainers. *Proc. 6$^{th}$ Int'nl Workshop on Program Compr.*, pages 180–187, 1998.

[105] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proc. 28$^{th}$ ICSE*, pages 905–908. IEEE CS Press, 2006.

[106] M. Smith, D. Weissa, P.Wilcox, and R.Dewer. *The Ophelia traceability layer*, pages 150–161. Franco Angeli, 2003. A. Cimitile and A. De Lucia and H. Gall (eds.).

[107] M. Storey and H. Mueller. Manipulating and documenting software structures using SHriMP views. In *Proc. 11$^{th}$ ICSM*, pages 275–284. IEEE CS Press, 1995.

[108] R. J. Turver and M. Munro. An early impact analysis technique for software maintenance. *J. of Softw. Maint. - Res. and Pract.*, 6(1):35–52, 1994.

[109] G. Venkatesh. The semantic approach to program slicing. *ACM SIGPLAN Notices*, 26(6):107–119, 1991.

[110] A. von Knethen and M. Grund. QuaTrace: A tool environment for (semi-) automatic impact analysis based on traces. In *Proc. 19$^{th}$ ICSM*, pages 246–255. IEEE CS Press, 2003.

[111] J. Weidl and H. Gall. Binding object models to source code. In *Proc. 22$^{nd}$ COMPSAC*, pages 26–31. IEEE CS Press, 1998.

[112] M. Weiser. Program slicing. *IEEE TSE*, 10(4):352–357, 1984.

[113] E. Wilde and D. Lowe. *Xpath, Xlink, Xpointer, and XML: A Practical Guide to Web Hyperlinking and Transclusion*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[114] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L. Pounds. A comparison of methods for locating features in legacy software. *J. of Syst. and Softw.*, 65(2):105–114, 2003.

[115] N. Wilde and M. C. Scully. Software reconnaissance: mapping program features to code. *J. of Softw. Maint. - Res. and Pract.*, 7(1):49–62, 1995.

[116] T. Wolf and A. H. Dutoit. Supporting Traceability in Distributed Software Development Projects. In *Proc. Int'nl Workshop on Distr. Softw. Develop.*, pages 111–124, 2005.

[117] W. Wong, S. Gokhale, J. Horgan, and K. Trivedi. Locating program features using execution slices. In *Proc. Symp. on Application-Specific Syst. and Softw. Eng. Tech.*, pages 194–203. IEEE CS Press, 1999.

[118] S. Yadla, J. Huffman Hayes, and A. Dekhtyar. Tracing requirements to defect reports: An application of information retrieval techniques. *Innovations in Syst. and Softw. Eng.: A NASA Journal*, 1(2):116–124, 2005.

[119] S. S. Yau and J. S. Collofello. Some stability measures for software maintenance. *IEEE TSE*, 6(6):545–552, 1980.

[120] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *Proc. 2$^{nd}$ COMPSAC*, pages 60–65. IEEE CS Press, 1978.

[121] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE TSE*, 30(9):574–586, 2004.

[122] M. Zalewski and S. Schupp. Change impact analysis for generic libraries. In *Proc. 22$^{nd}$ ICSM*, pages 35–44, 2006.

[123] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE TSE*, 31(6):429–445, 2005.

[124] A. Zisman, G. Spanoudakis, E. Perez-Minana, and P. Krause. Tracing software requirements artifacts. In *Proc. Int'nl Conf. on Softw. Eng. Res. and Pract.*, pages 448–455. CSREA Press, 2003.

[125] X. Zou, R. Settimi, and J. Cleland-Huang. Term-based enhancement factors for improving automated requirement trace retrieval. In *Proc. Int'nl Symp. on Grand Challenges in Traceability*, pages 40–45. ACM Press, 2007.