Reconstructing Requirements Coverage Views from Design and Test using Traceability Recovery via LSI

Marco Lormans
Delft Univ. of Technology
M.Lormans@ewi.tudelft.nl

Arie van Deursen
CWI and Delft Univ. of Technology
Arie.van.Deursen@cwi.nl

ABSTRACT

Requirements coverage views can help validate that all requirements are implemented in the system. This is not a trivial process. In this paper we present a method for generating requirements coverage views. To do this, information needs to be gathered from multiple sources in the development process. A traceability model defines the development work products and links that are required to generate the coverage views. Retrieving this information is done by using Latent Semantic Indexing (LSI). The method is applied in a lab study, Pacman, of which the preliminary results are also presented in this paper.

Keywords

Requirements Management, Traceability Recovery, Traceability Models, Requirements Views

1. INTRODUCTION

A requirement coverage view provides insight in the status of a requirement in a software development project. For example, coverage views include whether or not a requirement is covered by an acceptance test, by a design artifact, by a system test, and so on.

Up to date requirement coverage views can provide a major asset for developers and project managers, offering them a way to monitor progress in terms of coverage percentages. Unfortunately, managing a requirements set in such a way that adequate coverage views can be obtained is very hard in practice [9]. Up to date coverage information requires that an up to date traceability matrix is maintained, establishing links between, for example, requirements, test cases, design decisions, etc. Implementing traceability in practice and keeping the traceability links consistent during development of the product is a time consuming, error-prone, and labor-intensive process demanding disciplined developers [4, 8, 11].

In this paper we seek to investigate to what extent relevant traceability links can be reverse engineered automatically from available documents and used for generating coverage views. As observed by Alexander, tools currently available do not support the feature of automatically recovering traceability links during the evolution of a product [1].

The route we will explore in this paper is whether information retrieval (IR) techniques can be used to automate this reconstruction process. In this paper we use Latent Semantic Indexing (LSI) for recovering the traceability links [6]. LSI is a promising information retrieval technique assuming there is a latent semantic structure for every document set. A subspace of the latent semantic structure is used for retrieving information and reconstructing traceability links.

We tried our method in a lab study, Pacman, used in a testing course at Delft University of Technology. Available documentation includes use cases, design decisions, acceptance test cases, as well as a Java implementation with Java unit tests. The Pacman case gives us the opportunity to explore all the possibilities of our method in a controlled environment. In this study we varied the different parameters of our analysis to come to a setting giving the best results.

The long term purpose of our research is to find out how much industry can benefit from the advantages of using LSI to track and trace the requirements and generate the related coverage views. A key question is which traceability links can be reconstructed using LSI and which coverage views can be generated with this information. Furthermore we offer an analysis why these links can be reconstructed and what the requirements are for documenting their development artifacts.

The remainder of this paper is organized as follows. In Section 2 we give an overview of what requirements coverage views are and discuss related work. In Section 3 we give an overview of traceability recovery. In Section 4 we present our method for reconstructing coverage views using LSI, after which we describe and discuss our case study in Sections 5. We conclude the paper by summarizing the open issues and offering suggestions for future research.

2. REQUIREMENTS COVERAGE VIEWS

The different perspectives on requirements are often captured in views. Views capture a subset of the whole system in order to reduce the complexity from a certain perspective. Nuseibeh *et al.* discuss the relationships between mul-

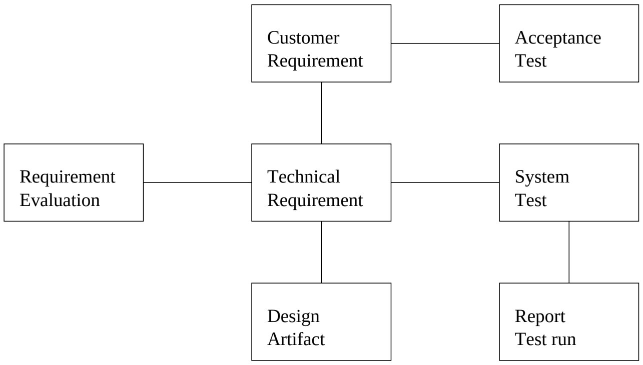

Figure 1: Traceability Model

tiple views of a requirements specification [16]. This work is based on the viewpoints framework presented by Finkelstein *et al.* in [7]. This framework primarily helps organizing and facilitating the viewpoints of different stakeholders.

Von Knethen also discuss view partitioning, but from a different perspective [20]: she considers views on the system, distinguishing, e.g., the static structure from the dynamic interactions in the system.

The application we have in mind is coverage monitoring. We are explicitly focusing on requirements coverage views. None of the discussed papers give an exact overview on what these should look like.

For reconstructing coverage views we need traceability to be implemented in the development environment. A traceability model defines the requirements entities together with their attributes and the traceability relations that are allowed to be set in the requirements management system.

An example of a traceability model relevant for coverage monitoring is shown in Figure 1. This model and the entities and dependencies contained in it reflect the way of working at a big industrial company. For example, it distinguishes between a *customer requirement* (cast in terms familiar by customer representatives) and *technical requirements* (cast in terms familiar by developers). Moreover, the model supports *evaluation* of requirements: after shipping the product, field studies are conducted in order to evaluate the working of the requirement in real life. The evaluation results are taken into account when shipping a new version of the product. This traceability model enables us to derive, for example, the following coverage information:

- **Identification coverage** indicates the links between customer requirements and technical requirements. The technical requirements specification defines the product that actually will be built. Every system requirement should have a link with at least one customer requirement, and vice versa.
- **Design coverage** ensures that the requirements in the system's requirements specification are addressed in the design. This view shows how well the design reflects the requirements. Note that the presence of a link does not mean that these requirements are also correctly implemented/designed. Having a requirements coverage of 100% after the design phase tells management that the system should have all functionality covered in the design as agreed in the contract.
- **Test case coverage**; A comparable situation applies to the requirements coverage in the test specifications. Most test specifications are created in the design phase in parallel with the design. This view shows how well the test specification reflects the requirements. Again this does not mean the functionality is correctly implemented. Having a coverage of 100% tells management that all functionality will be tested in the test phase.
- **Test pass coverage**; A system test is an internal test, often called factory test, to check if the system is working correctly. If all system tests pass, the development team can show the customer that all functionality is implemented and that all functionality is working correctly as agreed. This view shows which requirements are tested and ready for the customer acceptance test.
- **Acceptance coverage**; The customer can approve the results by means of the final acceptance test. This view shows which requirements are accepted by the customer and are ready for release.
- **Evaluation coverage**; After delivery, the evaluation coverage view indicates which requirements have been evaluated and are suitable for reuse in ongoing and future projects.

According to Costello *et al.* requirements coverage is defined as: *The number of requirements that trace consistently to the next level up or down* [5]. Costello *et al.* originally defined this metric for requirement to requirement coverage. As this definition is very general, it is also suitable for requirement to design coverage and requirement to test case coverage. In its simplest definition we can claim that if, for example, a link to a design artifact exists and this links is correct, the requirement is covered in the design. We take the percentage of the total number of requirements. This results in the following definition for coverage in design:

$$cov_d = \frac{|req_d|}{|req_{tot}|}$$

Cov_d represents the coverage in the design, req_d the number of requirements traced consistently by design artifacts and req_{tot} the total number of requirements. The same metric can be used for all the previous defined coverage views.

An example of a coverage view for an individual requirement can be one that shows its relations to other development artifacts (such as design artifacts and test cases) including its coverage status. Another example is a list of multiple requirements focused on a specific link, e.g. the link to design.

3. TRACEABILITY RECOVERY

To reconstruct coverage views from project documentation we need some traceability support. Several traceability recovery techniques already exist each covering different traceability issues during the development life-cycle.

Antoniol *et al.* use information retrieval (IR) methods to recover the traceability relations from C++ code onto manual pages and from Java code to requirements [2]. Marcus and Maletic use Latent Semantic Indexing (LSI) for recovering the traceability relations between source code and documentation [15]. For recovering traceability links between requirements, IR methods can also be used [10, 17]. Settimi *et al.* discuss the retrieval of links between requirements, code and UML models [19]. They used an IR technique based on a Vector Space Model. The results showed limited succes, but the use of IR does provide a reduction of time for maintaining traceability.

De Lucia *et al.* present an artifact management system, which has been extended with traceability recovery features [12, 13]. This system manages all different artifacts produced during development such as requirements, design artifacts, test specifications, and source code modules. De Lucia *et al.* also use LSI for recovering the traceability links. However presenting the results in specific coverage views is not supported.

Previous work mostly discusses the explicit relation between source code and documentation or the explicit relations between requirements on different levels of abstraction. In this paper we use traceability recovery already during the development of the system and more specific to support the future integration phase. Besides coverage views, it should also contribute to advances in systematic reuse of requirements, impact analysis, and conformance analysis.

The information retrieval technique we use for reconstructing the requirements traceability links is Latent Semantic Indexing (LSI) [6]. LSI is based on the Vector Space Model and assumes that there is some underlying or latent structure in word usage for every document set. This is particularly caused by classical IR issues as *synonymy* and *polysemy*. LSI uses statistical techniques to estimate this latent structure. A description of terms, documents and user queries based on the underlying latent semantic structure is used for representing and retrieving information. In this way LSI partially overcomes some of the deficiencies of assuming independence of words, and provides a way of dealing with synonymy automatically.

Another advantage is that it does not rely on a predefined vocabulary or grammar for the documentation. This allows the method to be applied without large amounts of preprocessing or manipulation of the input, which drastically reduces the costs of traceability link recovery [12, 14].

4. APPROACH

The long term objective of our work is an approach that supports big industrial companies in managing requirements throughout the life cycle of, for example, consumer electronics products or document systems such as copiers. Such products need to fulfil hundreds or thousands of requirements. Furthermore, these requirements can change over time when new product versions are created and shipped.

Our focus is on reconstructing *coverage views*, i.e., views on the requirements set that can be used to monitor the progress in requirements development, design, and testing. The steps involved in our approach include:

1. Defining the underlying traceability model;
2. Identifying the concepts from the traceability model in the available set of documents;
3. Preprocessing the documents to support automated analysis;
4. Reconstructing traceability links by means of LSI;
5. Selecting the statistically relevant links and reducing the full set of links identified;
6. Generating coverage views by capturing the relevant information.

Below we elaborate each of these steps. Moreover, we discuss how we can evaluate the results of our approach.

4.1 Traceability Models for Coverage Views

Traceability relations establish links between requirements on the one hand and various types of development work products on the other. A traceability model defines the work products and the types of links that are permitted within the development process.

The choice of traceability model mainly depends on the purposes for which it is to be used. For example, Ramesh and Jarke [18] discuss a range of different traceability models. Other examples of reference models can be found in [14, 20]. This step results in a traceabilty model as presented in Figure 1.

4.2 Concept Identification

Every concept contained in the traceability model should be uniquely identified in the available documentation. Since the documents are typically semi-structured (typically being just MS Word files), this requires a certain amount of manual processing. The more systematically the documents are organized (for example through the use of templates), the easier it is to make this structure explicit and identify the texts for each of the entities from the traceability model.

In general, identifying the relevant text will be somewhat easier for requirements, use cases, or test cases, which in most development approaches are tagged with a unique identifier. It is generally not so clear how design decisions should be documented and identified. Key decisions are often captured in (UML) diagrams. Here, we encounter the well known problem of establishing traceability relations between requirements and design [19].

4.3 Text Preprocessing

After defining the entities and the texts belonging to each of them, some pre-processing of these texts is needed. The first is extracting the texts from the original documents, bringing them in the (plain text) input format suitable for further automated processing. This often a manual task. The next step is to conduct typical IR steps such as lexical analysis, stop word elimination, stemming, index-term selection, and index construction. For this we use the Term-Document Matrix Generation toolbox called TMG [21].

4.4 Link Reconstruction Using LSI

After generating the term-by-document matrix we can reconstruct the traceability link using LSI. First of all, it creates the rank-k model on the basis of which similarities between documents can be determined. Here we need to choose the number for k. Secondly, for every link type in the traceability model (for example tracing requirements to designs) a similarity matrix is created containing the similarities between all elements (for example between every requirement and design artifact). The result of our LSI analysis is a similarity matrix containing the recovered links, represented as their similarity measures. This allows you to judge the quality for every recovered link.

4.5 Link Selection

Once the similarity matrix is available a choice has to be made if the similarity number is indeed a traceability link or not. There are different approaches for doing this [12]. We use a combination of *constant threshold* and *variable threshold* [12, 15].

This approach defines a constant threshold c and a variable threshold ε depending on the minimum and maximum similarity measure. First all the measurements are compared with the constant threshold (a commonly used threshold is $c = 0.7$) to indicate if there is any similarity. If all measures are smaller than c there are no links at all. If there are measures greater than the constant threshold we take the variable threshold ε for choosing the traceability links. With the variable threshold a similarity interval is defined by the minimum and maximum similarity measure. This way it is possible to take, for example, the best 20% of all similarity measures. In this case the 20% best measures are traceability links.

For measuring the performance of the traceability reconstruction we use two well-known IR metrics: *recall* and *precision* [12]. Both parameters c and ε influence the performance indicators recall and precision as will be shown in the case study.

4.6 Generating Coverage Views

The final step is to use the reconstructed traceability links to obtain coverage views. Given the presence of a link, the status of a requirement can be appropriately set. The recovered traceability matrix is used to calculate the coverage metric defined in Section 2. Currently, for every link defined in the traceability model we calculate the percentage of all requirements covered by the specific link. So we get a list of requirements coverage percentages in the design, test cases and so on.

5. CASE STUDY: PACMAN

In this section we will discuss our preliminary results from the a case study we executed at Delft University of Technology. This is a lab experiment and is used by students in a lab course for testing object oriented software following Binder's testing approach [3]. The system at hand is a simple version of the well-known Pacman game. An initial implementation for the system is given, and students are expected to extend the test suite according to Binder's patterns, and enhance the system with additional features (which they should test as well). We will only shortly discuss the configuration and try to focus on the observations and findings.

5.1 Case Configuration

In this case study three main documents are provided: a requirements specification, a design document and a test specification. Along with these documents a traceability matrix is provided, which captures the correct traceability links. In this study we focus on the links between requirements and design, and requirements and test cases.

These specifications were in plain text already and could be passed directly as input to the TMG toolbox. For the requirements specification the use cases are chosen as requirements entities. The design is event-oriented so the design artifacts are specified according to the events in the system. Finally, every test case is copied as test case entity for our analysis. In total there are 10 requirement entities (use cases), 19 design artifacts and 17 test cases.

As corpus, the collection of all documents was used, including the implementation. This resulted in a corpus of almost 1200 terms. Furthermore, the values for k, c and ε need to be defined. For c we took 0.7. The other two values we varied between 10% and 50% to get an impression of the impact of these values.

5.2 Results

The most promising results were obtained with k set to 20%, and ε at either 20% or 30%. Both the traceability matrix for the use case to test case mapping and the use case to design artifact mapping were computed. As an example, the similarities between use cases and test cases as computed via LSI are shown in Table 1. The figures in this table were computed with an ε of 30%.

The bold numbers in the table correspond to entries that were marked as a correct link by a domain expert (the author of the Pacman implementation). The X cell at the bottom right denotes a link that was marked by the expert, but not found by our LSI analysis. Thus, only one out of 17 links was missed, giving a recall of 94%. Finally, this results in a coverage of 90%.

The recall and precision for our experiments are shown in Table 2. For both types of traceability links, recall, precision and coverage are given for ε values of 20% and 30%. In both cases, recall increases as the ε is larger and consequently so does the coverage. Remarkable is that for the use case to test traceability links, the precision does not get lower with an increase in ε.

5.3 Observations

The relatively small size of the Pacman application and our familiarity with the requirements, test strategy, and implementation allows us to offer an explanation for the traceability matrices obtained.

One of the reasons that LSI finds more than the original matrix (and the resulting lack of precision), is that some of the use cases are similar themselves. For example, use cases UC7 and UC9 are textually fairly similar, dealing with restarting

		UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	GUI
		start	move	eat	p-die	m-move	m-die	suspend	exit	restart	
TC1	start	**0.84**						0.78	0.68	0.77	0.69
TC2a	empty		**0.77**			0.77					
TC2b	border		**0.82**			0.83					
TC2c	wall		**0.79**			0.82					
TC3a	food			**0.71**							
TC3b	win			**0.80**							
TC4	p-die				**0.76**		0.78	0.72		0.80	
TC5a	m-move					**0.72**					
TC5b	border		0.77			**0.81**					
TC5c	wall					**0.80**					
TC5d	food					**0.79**					
TC6	m-die		0.84	0.76	0.83	0.78	**0.86**	0.80		0.85	
TC7	suspend	0.71						**0.73**	0.72	0.79	0.71
TC8	exit								**0.74**		0.72
TC9a	restart-d							0.71		**0.85**	0.70
TC9b	restart-w							0.73		**0.86**	0.70
TC10	gui			0.70					0.77		X

Table 1: Traceability Matrix for the Pacman Case Study

after a suspend event (UC7) versus restarting after a game over event (UC9). A consequence of this similarity is that the same test cases are associated with both of them. Likewise, the constraints for moving the player (UC2) and those for moving a monster (UC5) are similar. The same can occur for test cases: TC4 (the player hits upon a monster) and TC6 (a monster hits upon the player) are very similar, causing associations with the same use cases.

The single link that was not found is the requirement for the graphical user interface and the corresponding test case. This requirement is not expressed as a use case, but explains the intended layout and behavior of the graphical user interface. As such, it mentions several of the Pacman game elements, which explains why it is linked to several test cases. The GUI test case is a description of what a person testing the GUI should look at in order to validate correct behavior. This was expressed as a series of statements like "inspect that ...", which in this case were not sufficiently similar to the GUI description.

Looking at the recall/precision figures in Table 2, we see that the recall and the precision are lower for the use case to design traceability matrix. The explanation here is that requirements-design traceability links are inherently harder to determine. In the design we deal with a model representing static (classes) and dynamic (state machine) behavior.

A final observation is that it was not easy to map static class structures to dynamic behavior as described in the use cases. Establishing links was a little easier for the design decisions related to the state machine. Both use cases and state transitions address dynamic aspects of the game, and were easier to connect. Most of the recall indeed comes from the design decisions related to the state machine. It could be that dynamic analysis may be a more suitable reconstruction technique here.

Note that these observations are in accordance with known limitations of the use of LSI for small case studies [10, 19].

Link type	ε	Recall	Precision
Use case to test	20%	0.71	0.38
	30%	0.94	0.37
Use case to design	20%	0.60	0.39
	30%	0.68	0.29

Table 2: Recall and precision for the reconstructed traceability matrices

6. OPEN ISSUES AND FUTURE WORK

The Pacman case study first of all shows that if the documents to be analyzed are setup according to a well-designed traceability structure, LSI is capable of reconstructing it. The other side of the medal is that if there is no structure (or only very little) in the underlying data, we cannot expect that LSI will find it. The implication of this is that the use of LSI-based traceability recovery for coverage view reconstruction cannot yet replace a systematic requirements management process. In such a process, the careful design of traceability relations should still play a key role. Our approach can help, however, in keeping the traceability links maintained for such a system.

We are also aware that the Pacman study is small for using IR techniques. The relatively small size of the Pacman texts, makes LSI more vulnerable to deviations due to spelling errors, synonyms, or alternative formulations. Still the results were promising. This case study was primarily intended to explore the possibilities. Future work should investigate what the minimal size of an industrial setting should be to use our approach as presented in this paper and how we can make our approach more robust with e.g. other technology such as dynamic analysis. Our industrial case study at Philips Applied Technologies is much bigger and will provide a large amount of data to investigate these issues.

In the Philips Applied Technologies case study we will focus on an extension of a consumer electronics product. This product has a long history all kept in a product database. Requirements but also other development artifacts are reused from previous products. During development a large number of requirements initially identified cannot be traced back to test cases or design documents: in a way they "get lost". This gets even worse when the system evolves over time. First ad-hoc attempts in two case studies showed that less than 10% of the total requirements can be recovered from the design and test documents. Furthermore, as the system evolves, new requirements are introduced that cannot be traced back to the requirements specifications and for that reason should not be there at all. Philips would like to monitor the development process using specific coverage views. These views should be enriched with more detail.

In this paper we attempt to reconstruct the design and test case view. In the end, these views should also include e.g. test pass and evaluation information. In other words, the view should be enriched and give and overview of the exact status of the requirement (or requirements) during development. One of our particular interests is how to capture hierarchically structured requirements. What are the consequences for our coverage views and can we automatically generate a coverage view, which includes all this information? Secondly, we need to investigate if LSI is able to retrieve this information. How does LSI dead with hierarchically structured requirements?

One of the things we also need to focus on is a more formal traceability model. If we observe various industrial settings many potential links can be identified. These need to be formalized and captured in the traceability model.

Finally, our findings showed that tracing requirements in test cases is easier than tracing requirements in design. One of the thinks we like to investigate further is if we can improve traceability from requirements to design via test cases? Test cases are mostly developed in parallel with the design and are closely related to the design; they often test a specific design artifact. This way we can indirectly retrieve information about requirements coverage in design.

7. REFERENCES

[1] I. Alexander. Towards automatic traceability in industrial practice. In *Proc. of the 1st Int. Workshop on Traceability*, pages 26–31, Edinburgh, UK, 2002.

[2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[4] S. Brinkkemper. Requirements engineering research the industry is and is not waiting for. In *Proc. of the 10th Int. Workshop on Requirements engineering: Foundation for Software Quality*, 2004.

[5] R. J. Costello and D.-B. Liu. Metrics for requirements engineering. *Journ. of Sys. and Softw.*, 29:39–63, 1995.

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

[7] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. Journal of Software Eng. and Knowledge Eng.*, 2(1):31–58, March 1992.

[8] O. Gotel and A. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the First IEEE Int. Conf. on Requirements Engineering*, pages 94–101, Colorado springs, April 1994.

[9] B. Graaf, M. Lormans, and H. Toetenel. Embedded software engineering: state of the practice. *IEEE Software*, 20(6):61–69, November–December 2003.

[10] J. H. Hayes, A. Dekhtyar, and J. Osborne. Improving requirements tracing via information retrieval. In *Proc. of the 11th IEEE Int. Conference on Requirements Engineering*, pages 138–147, Washington, DC, USA, 2003. IEEE Computer Society.

[11] M. Lormans, H. van Dijk, A. van Deursen, E. Nöcker, and A. de Zeeuw. Managing evolving requirements in an outsoucring context: An industrial experience report. In *Proc. of the Int. Workshop on Principles of Software Evolution*, Kyoto, Japan, 2004.

[12] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Enhancing an artefact management system with traceability recovery features. In *Proc. of the 20th IEEE Int. Conference on Software Maintenance*, pages 306 – 315. IEEE Computer Society, 2004.

[13] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Adams re-trace: A traceability recovery tool. In *Proc. of the 9th European Conference on Software Maintenance and Reengineering*, pages 32–41. IEEE Computer Society, March 2005.

[14] J. I. Maletic, E. V. Munson, A. Marcus, and T. N. Nguyen. Using a hypertext model for traceability link conformance analysis. In *Proc. of the 2nd Int. Work. on Traceability in Emerging Forms of Software Engineering*, pages 47–54, Montreal, Canada, 2003.

[15] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proc. of the 25th Int. Conference on Software Engineering*, pages 125–135, Washington, DC, USA, 2003. IEEE Computer Society.

[16] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Trans. Softw. Eng.*, 20(10):760–773, 1994.

[17] J. N. och Dag, V. Gervasi, S. Brinkkemper, and B. Regnell. A linguistic-engineering approach to large-scale requirements management. *IEEE Softw.*, 22(1):32–39, 2005.

[18] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, 2001.

[19] R. Settimi, J. Cleland-Huang, O. B. Khadra, J. Mody, W. Lukasik, and C. DePalma. Supporting software evolution through dynamically retrieving traces to UML artifacts. In *Proc of the 7th Int. Workshop on Principles of Software Evolution*, pages 49–54, Washington, DC, USA, 2004. IEEE Computer Society.

[20] A. von Knethen. A trace model for system requirements changes on embedded systems. In *Proc. of the 4th Int. Workshop on Principles of Softw. Evolution*, pages 17–26, New York, NY, USA, 2001. ACM Press.

[21] D. Zeimpekis and E. Gallopoulos. Design of a matlab toolbox for term-document matrix generation. In *Proc. of Workshop on Clustering High Dimensional Data and its Applications*, pages 38–48, Newport Beach, California, April 2005.