

# Traceability ReARMed

## Abstract

*Traceability links connect artifacts in software engineering models to allow tracing in a variety of use cases. Common to any of these use cases is that one can easily find related artifacts by following these links. Traceability links can significantly reduce the risk and cost of change in a software development project. However, finding, creating and maintaining these links is costly in most cases. In any real-world project of significant size the creation and maintenance of traceability links requires tool support.*

*In this paper, we propose a novel approach to support the automation of traceability link recovery based on Association Rule Mining and operation-based change tracking. Traceability link recovery is the activity of finding missing or lost traceability links. Our approach automatically generates a list of candidate links based on the project history along with a score of support and confidence for every candidate link. We transformed the data from an operation-based change tracking system to sets of frequent items, which serve as input for Association Rule Mining (ARM). We applied our approach to data from a software development project with more than 40 developers and assessed the quality of the candidate links in interviews.*

## 1 Motivation

Traceability is an important property of any software model and can be defined as follows: “Software artifact traceability is the ability to describe and follow the life of an artifact (requirements, code, tests, models, reports, plans, etc.) developed during the software lifecycle in both forward and backward directions (e.g., from requirements to the modules of the software architecture and the code components implementing them and vice-versa)” [12] [18]. Missing support for traceability has been recognized as a major cause for project overruns and failures [7] [16]. Due to the iterative nature of software development, the manual detection and maintenance of traceability links has to be performed frequently and is costly [4]. The need for tools to support finding and maintaining traceability links

is widely recognized in academia [9] [25] [21] [22]. Existing approaches for recovering traceability links are based on content and rely on the hypothesis that related artifacts are also related in content. Traceability links can therefore only be found if the two artifacts that are to be linked have similar content. This is a major drawback and either results in many false positives if the content similarity threshold is low, or in many false negatives if it is high. During the case study presented in section 5, we also observed that related artifacts must not be similar in their content at all, especially if the content is sparse and the model is still under development. De Lucia et al. [18] explicitly suggest to solve the limitations of content-based approaches by complementing them with other approaches.

In this paper, we propose a novel approach to traceability link recovery called ReARM (**R**ecovery of traceability links based on **A**ssociation **R**ule **M**ining). ReARM does not rely on content similarity, but on the history of a software model, overcoming the limitations of content-based approaches [18]. Especially in models under development, where content may still be sparse, content-based similarity is not an appropriate indicator for related artifacts. By recovering links from the previous change and navigation behavior on the model, the candidate links are restricted to relevant parts of the model and to links that would have been of use in the past. Our approach is also applicable to unified models, which incorporate a broader range of software engineering artifacts than previous software models. Further, ReARM can be used in addition to existing content-based approaches, completing their results with links that are not related in content but according to their change history.

## 2 Related Work

For tool-supported traceability link recovery, there are two types of approaches in general, content-based and non content-based ones. Several content-based approaches make use of methods from Information Retrieval (IR) such as Latent Semantic Indexing (LSI), deriving traceability link candidates from content similarity. Non content-based approaches often use data mining techniques, such as Association Rule Mining (ARM), to discover traceability links

symbolically.

**Content-based Approaches** The majority of existing approaches is based on content. Several content-based approaches employ basic matching techniques as done by Murphy et al. [21], where regular expressions and naming conventions are leveraged to map the developer’s model to the source model, using the “reflection model technique”. The tool QuaTrace by Maletic et al. [25] integrates the commercial tools RequisitePro and Rhapsody to provide traceable requirements using guidelines during document creation and name matching. Zisman et al. [29] also use syntactical matching of related terms in requirements and in the source code of the system. All these approaches depend on naming conventions and adherence to guidelines.

As mentioned before, many content-based approaches are based on information retrieval methods. According to Antoniol et al. [2], the probabilistic model and the vector space model from information retrieval are both suitable for traceability recovery and show similar performance in this field. These methods require a prior stemming step, i.e. the reduction of words in a text to their corresponding stems to make comparisons easier. Settimi et al. [24] utilize the vector space model to trace requirements to UML artifacts and code. In [13], Hayes et al. introduce the RETRO tool which utilizes an improved vector space model technique as well as latent semantic indexing (as an extension of the vector space model). David proposes a recommendation system, which is based on a recurrent neural network as well as on LSI, to predict likely further changes – given a set of already changed software artifacts [6].

Cleland-Huang et al. [5] introduce three strategies for improving probabilistic retrieval algorithms, i.e. hierarchical enhancement, clustering enhancement and graph pruning enhancement. Richardson and Green [22] use a method called surface traceability, in which small perturbations are made and the corresponding changes are observed to automatically discover traceability links between the specification and parts of a program – generated from this specification using program synthesis. While this technique is interesting for generated programs, it was not developed for traceability discovery in manually coded programs.

Maletic and Marcus [19] make use of latent semantic indexing, showing that latent semantic indexing performs better than vector space or probabilistic models and does not rely on stemming either. Maletic et al. [20] extend the latent semantic indexing approach by creating a formal hypertext model for link recovery and maintenance of the conformance of the links over time. Lormans and van Deursen [17] reconstruct links between requirements on the one side and design artifacts and test case specifications on the other side also using latent semantic indexing. They discuss two possible link selection strategies, which include

to apply a 1-dimensional as well as a 2-dimensional filter to the term-document matrix. De Lucia et al. [18] created an artifact management system with traceability recovery based on latent semantic indexing called ADAMS. They conclude that in order to go beyond the limitations of information retrieval methods, a combination with syntax-based approaches would be necessary. Summing up the existing approaches in the field of traceability link discovery, content-based approaches mainly use information retrieval methods based on the vector space model such as LSI. These techniques rely on static similarity between different development artifacts (entity-centric). Sparse or missing artifact contents imply a need for non content-based approaches such as syntax-based analysis or association rule mining. These methods often address traceability from an activity-centric point of view, discovering traceability links from the traces of different development activities.

**Non Content-based Approaches** The CAESAR approach developed by Gall et al. [10] attempts to measure the coupling of multiple releases of a system – based on empirical analysis. Using a product release database, they identify change patterns using change sequence analysis and change report analysis and employ these patterns to detect coupling between subsystems. Egyed and Grünbacher [9] utilize a graph of nodes, which can be built by observing the classes and methods used while testing a scenario. Trace analysis is then performed by iteratively manipulating this graph. Gall et al. [11] exploit CVS (Concurrent Versions System) data by quantitative analysis, change sequence analysis, and relation analysis to analyze software evolution and to display change behavior. CVS data is also the basis for the work of Ying et al. [27], who utilize association rule mining to determine change patterns from the change history of the CVS repository. They exploit this information to recommend possibly relevant artifacts to the programmer. ROSE, as presented in Zimmermann et al. [28], also makes use of association rule mining on a CVS repository to predict future changes, to reveal the coupling of entities, and to prevent errors occurring due to the incompleteness of changes. Both approaches operate on the source code only.

Non-content based approaches traditionally determine change patterns based on the changes to a CVS-like repository. Change-impact mining on a unified model using a broader range of software engineering artifacts, as described in section 3.1, can help to improve these approaches.

### 3 Prerequisites of the ReARM Approach

In this section we will introduce prerequisites for our approach. The traceability links we recover are instances of a unified model that combines models from different domains

such as UML models, rationale models, or schedules. In section 3.1, we describe the idea of a unified model and define its terminology [26]. Since we recover traceability links from the history of the models, our approach is based on Software Configuration Management (SCM), more specifically, on a operation-based versioning system (section 3.2, [15]). Finally, we introduce association rule mining as the data mining technique used to process the history data (section 3.3).

### 3.1 Unified Model

In software engineering, a model can describe the system under development on different levels of abstraction, at different points in time, and from different points of view. Typically, in a software development life-cycle, many models of the system under development are produced. These models range from requirements models to class models of the entities involved in the system. However, models in a software development project may also build an abstraction of the software development project itself in terms of the organization, resources, or schedules. Many management artifacts fall into this category, such as work breakdown structures or organizational charts. We therefore define the term *system model* as the entirety of models describing the system under development, while the *project model* is the entirety of models describing the development project itself.

Models define model elements and links, in the terminology of a unified model they represent an aggregation of model elements and model links. Model elements are nodes that have a certain number of attributes of a certain type. For example, in a UML class model, there are class nodes including but not limited to the attributes name, is-Abstract, and package. The requirements model contains requirement nodes with attributes such as stakeholder or priority. Model links define the connections that are valid among nodes. In the UML class model example, links include association, aggregation, and composition links, in the requirements model links might specify a refiningRequirements relation. Essentially, all instances of models are graphs with instances of model elements as nodes and instances of links as edges.

Since several models can describe the same system under development, it is often useful to add additional links connecting elements of these different models. For example, it can be essential to link a requirement from a requirement model with a use case from a use case model to express that the use case is describing the requirement in greater detail. A unified model defines models with their model elements and links among the model elements from the same or different models.

### 3.2 Operation-based Change Tracking

Commonly instances of a model are stored in a Software Configuration Management (SCM) system. Our approach is based on the history provided by an SCM system, which serves as input for the association rule mining algorithm. For the ReARM approach, we need to analyze the changes that were performed on the granularity of model elements and links. Most existing SCM systems however provide data about the evolution of a model on the granularity of files. We therefore rely on the SCM system presented in [15] and implemented in Sysiphus [8]. This SCM system works in an operation-based way and provides change information on the granularity of model elements and their attributes. Each change is called an operation and describes the attribute that was changed, the respective model element, and the old and new value of the attribute. All model elements also contain reader information, that is information about the time a user has last read the element, if at all. Since reading a model element will update its reader information, reading a model element is also recorded as a change which we will exploit in the following.



Figure 1. Instance of a version graph.

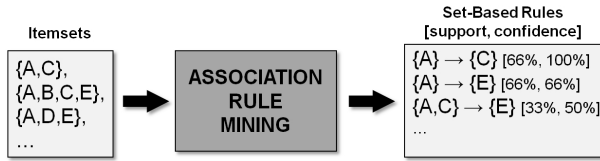
The SCM system records the history of an instance of a unified model in a version graph. Figure 1 shows an instance of a version graph. Each instance of Version represents a project state, having a predecessor and a successor Version instance. Modifying version *version1* creates version *version2*. The modifications on the model that occurred in *version1* transforming it into *version2* are represented by the instance *changesFrom1To2* of the association class ChangePackage. From a technical point of view, we say applying *changesFrom1To2* to *version1* results in *version2*. The ChangePackage *changesFrom1To2* is a composite of three ModelOperation instances. The ModelOperation *linkAtoB* is an operation that was performed on the model between *version1* and *version2*. It describes the creation of a link between two model elements A and B, where A is a requirement and B is a use case.

On the basis of this change tracking infrastructure, the PAUSE framework [14] was constructed. It provides access to project iterators returning the project state and the

changes applied to the project on any day of the project life-cycle.

### 3.3 Association Rule Mining

Association rule mining (ARM) is a data mining technique usually applied to large databases of sets of co-occurring items. The technique was first introduced by Argawal, Imielinski, and Swami [1], who used it for market basket analysis. Its goal is to discover buying patterns of the form “85% of the purchases including diapers also included beer”. Such probabilistic rules can then be used for decision support systems regarding price promotion or store layout. Figure 2 shows the functionality of ARM from a blackbox perspective, receiving itemsets as input and providing discovered rules as output.



**Figure 2.** Blackbox view of association rule mining with generic input and output representation that is applicable to many different domains and their problems.

Recently, association rule mining has also been applied to software engineering. One contribution was made by Zimmermann and his colleagues, who mined version histories to guide software changes [28]. When a developer works on a programming task, relevant software artifacts such as documents, source code, multimedia files, etc. that might require corresponding changes, can be proactively recommended: “*Programmers who changed these artifacts also changed ...*”.

In this paper, we also use an association rule mining technique based on the apriori algorithm [23]. We define the fundamental concepts of association rule mining [3] by the following terminology.

- *Items*  $I = \{i_1, i_2, \dots, i_m\}$ : The set of all existing items each represented by a unique symbol.
- *Itemset*  $X \subseteq I$ : A subset of the globally available items, e.g. a shopping cart of goods to be bought. For facilitating the mining process, the items are ordered lexicographically,  $(x_1, x_2, \dots, x_n)$  with  $x_1 \leq x_2 \leq \dots \leq x_n$ . A  $k$ -itemset is an itemset containing  $k$  items  $x_1, \dots, x_k$ .
- *Transaction*  $T = (tid, X)$ : Each transaction is a 2-tuple consisting of a  $tid$ , which is simply an identifier in form of a natural number, and an itemset  $X$ .

- *Database*  $D = \{T_1, T_2, \dots, T_n\}$ : A database is the collection of transactions, which were captured and stored in a transaction table  $\langle tid, X \rangle$ .
- The *cover* of an itemset  $X$  is a set of identifiers, denoting the transactions that contain  $X$ :  $cover(X) := \{tid | (tid, Y) \in D, Y \subseteq X\}$ .
- The *frequency* of an itemset  $X$  in the database  $D$  is the probability of its occurrence within a transaction  $T \in D$ :  $freq(X) = \frac{support(X)}{|D|} = \mathbb{P}(X)$ , where  $support(X) := |cover(X)|$ .
- Itemsets  $X \in D$  that satisfy a given frequency threshold  $t$ ,  $freq(X) \geq t \in [0, 1]$ , are of special interest.

The concrete values of these terms are typically measured by counting the entries of a transaction database, which represents the dataset to be analyzed. ARM works upon unordered sets and not upon ordered sequences, which is a fundamental premise and constraint of this technique. Thus order-sensitive sequences such as words of a grammar cannot be represented without loss of information.

The ARM process consists of two subsequent steps; first, all frequent itemsets in the database have to be discovered. In the second step, the actual association rules are derived from the set of frequent itemsets: **1. Mining of Frequent Itemsets**: There are  $\binom{|I|}{k}$ -many  $k$ -itemsets that all have to be checked for minimum support in case of a naive approach to itemset mining. Thus the worst case costs sum up to  $O(\sum_{k=1}^{|I|} \binom{|I|}{k}) = O(2^{|I|})$ ; **2. Association Rule Mining**:

- An association rule is an implication of the form  $X \Rightarrow Y, X \cap Y = \emptyset$ .  $X$  is the rule’s *body*,  $Y$  is the *head*.
- The *support* of an association rule  $A \equiv X \Rightarrow Y$  is computed based on the support of (frequent) itemsets:  $supp(A) = supp(X \cup Y)$ .
- The *confidence* of an association rule  $A \equiv X \Rightarrow Y$  is defined as:  $conf(A) = \frac{supp(X \cup Y)}{supp(X)}$ .

The confidence can also be interpreted as *conditional probability* of the rule’s head:  $\mathbb{P}(Y|X) = \frac{\mathbb{P}(Y, X)}{\mathbb{P}(X)}, \mathbb{P}(X) \neq 0$ . The desired minimal confidence is chosen by domain experts before starting the frequent itemset mining.

The association rule mining step requires a set of frequent itemsets as input, which is the output of step 1. Based on the discovered  $k$ -itemsets, which are frequent in the underlying database, association rules  $A$  of the form  $A := (Y \Rightarrow X - Y)$  are computed. Only rules with the desired minimal confidence are considered. The confidence  $conf(A)$  can be calculated  $\forall Y \subset X$  in the following simple way:  $conf(A) = \frac{supp(Y \cup (X \setminus Y))}{supp(Y)} = \frac{supp(X)}{supp(Y)}$ .

**Apriori Algorithm** To solve the frequent itemset problem in a more efficient way than checking each  $k$ -itemset  $X = \{(x_1, x_2, \dots, x_k)\}$ ,  $x_i \in \{i_1, i_2, \dots, i_m\}$ ,  $i \in \{1, 2, \dots, k\}$  regarding its support (a total of  $2^m - 1$  evaluations), a dynamic programming approach is carried out. This technique is known as the apriori algorithm for ARM, which exploits a *monotony property*. The monotony property states that all supersets  $X'$  of an itemset  $X$ ,  $X \subset X'$ , cannot be frequent, if  $X$  is not frequent. This property narrows the search space significantly and thus speeds up the computation of frequent itemsets. The constrained search space corresponds to the set of *candidate itemsets*, whose support has to be computed. Since the candidate generation can be additionally pruned by discarding those candidates not contained in the transaction database, the whole ARM algorithm is very fast (magnitude of several seconds) for up to several hundred thousands of itemsets, which additionally qualifies this technique for traceability link discovery.

#### 4 Traceability Link Discovery Approach

In the ReARM approach, we apply association rule mining (ARM) to the history and therefore also to the user navigation traces of software models to recover traceability links. Our main contribution is the combination of ARM and operation-based change tracking as well as their application to recover traceability links. Also we apply ReARM to a broad set of artifacts from a unified model (cf. section 3.1). We only rely on the history of the models not on the content of any model element, overcoming the limitations of content-based approaches [18]. Especially in models under development, where content may still be sparse, content-based similarity is not an appropriate indicator for related artifacts. By recovering links from the previous change and navigation behavior on the model, the candidate links are restricted to relevant parts of the model and to links that would have been of use in the past. In the following, we describe our approach to recover traceability links, presenting a short overview of the recovery process and then describing the details of each step. Figure 3 shows an overview over the process that consists of three major steps: session mining, read-read partitioning, and association rule mining.

As outlined in section 3.2, the history of a model instance is recorded in change packages. Our first step is to cluster the change packages into sessions per author. A session is a list of change packages from one author, where the time between two consecutive change packages remains below a given threshold, the session timeout. In a further step, certain changes are filtered from the sessions, e.g. those that relate to deleted elements and thus are useless. Each session represents a set of model operations, which is partitioned according to a read-read heuristic in the second step.

Finally, the association rule mining procedure results in a set of rules. Each rule is given by a set of model elements on the left hand side and a set of model elements on the right hand side of the rule. Each rule represents a candidate for a traceability link, which is annotated with a confidence and a support value.



**Figure 3.** Coarse-grained work flow of the ReARM traceability link discovery approach.

**Step 1: Session Mining** The history of a model instance consists of a list of change packages representing all changes that ever occurred on this model instance. Each change package is a list of model operations. Each change package was created by exactly one author and was committed at a given point in time. We define a *session* as a list of model operations from one author so that the period of time in between two succeeding change packages is within a given *session timeout*. This way, we are able to transform the list of change packages into a list of sessions. If two commits from one author occur within a few minutes or even seconds, we observed that the authors are working on similar modeling tasks in most cases. In an ideal scenario, a commit would be equal to a session and each author would commit only after and also immediately after completing one modeling task. The session timeout represents a degree of freedom in this step, thus we vary this parameter in our evaluation.

**Step 2: Read-Read Partitioning** In the second step, we need to filter the model operations in the sessions to avoid some unwanted results. As we are only able to find traceability links between existing elements, we filter all model operations referring to deleted elements. Additionally, we remove changes that refer to administrative model elements, such as project configuration data. On every read of a model element, its reader information is updated and recorded as a change. Based on the hypothesis that users trace artifacts even if they are not connected by a link, we filter the changes to reader information changes. Thereby, we model the navigation of a traceability link by frequent and sequential reader information changes on model elements to be linked. As input for ARM, we use a set of tuples of model elements. Each pair of succeeding reader information changes on two model elements that are not yet linked represents such a tuple within the session timeframe. In the case a traceability link is absent, the user may take several approaches to find the correct element, therefore we also include transitive links up to a certain depth (transitivity-

depth). In other words, for a fixed model element, we generate  $n$  pairs with the  $n$  succeeding elements that were read – not only with the direct successor. As result of step 2, we obtain a set of tuples of model elements that were read in sequence for each session obtained from step 1.

**Step 3: Association Rule Mining** In the final preprocessing step, the two model elements from each tuple are joined to get one itemset. The collection of all of these itemsets is fed into the ARM mechanism as a set of sets. Before starting the mining process, the two parameters support and confidence are chosen, e.g.  $minSupp = 0.015$  and  $minConf = 0.7$ , which means that the items that take part in an association rule occur in at least 1.5% of all itemsets. The more itemsets are contained in the underlying database, the lower the  $minSupp$  value has to be chosen, since otherwise very few association rules would be found. Now consider the association rule  $A := (X \Rightarrow Y)$ , where  $supp(X \cup Y) \geq minSupp$  (frequent itemset). The value  $minConf$  is the minimal fraction of itemsets (here 70%) that also contain the elements of itemset  $Y$ , given that they already contain those of itemset  $X$ , more formally,  $conf(A) = \frac{supp(X \cup Y)}{supp(X)} \geq minConf$ .

Finally, the ARM algorithm returns a set of rules  $\{A_1, \dots, A_n\}$ , each of them fulfilling the  $minConf$  constraint. Since our input only contained tuples the cardinality is two for each  $A_i$ . We only keep those rules, which point from the first element to the second element in any tuple. In other words, we discard those rules, whose left hand side (body) as well as right hand side (head) are never the first or second element in a tuple respectively. This way, we obtain rules with the following meaning: Reading the model element on the left hand side often results in reading the model element on the right hand side. Each rule represents one traceability link candidate. Each candidate is annotated with a support and a confidence value from ARM (cf. section 3.3). In the following evaluation, we present the quantitative results of the ReARM approach applied in an industrial case study.

## 5 Evaluation

We evaluated the ReARM approach in a case study of a project named DOLLI (Distributed Online Logistics and Location Infrastructure) at a major European Airport. The objective of the DOLLI project was to improve the airport’s existing tracking and locating capabilities and to integrate all available location data into a central database. Luggage tracking and dispatching of service personal as well as a 3D visualization of the aggregated data were developed. More than 40 developers worked on the project for about five months. All modeling was performed in the Sysiphus CASE tool [8]. This resulted in a comprehensive project model consisting of about 15.000 model elements and a his-

tory of over 53.000 versions.

Before presenting the quantitative results of our evaluation, we give a concrete example of a link candidate generated by ReARM based on the history of the DOLLI project. One of the traceability link types within the system model points from use cases to UML classes and to UML class diagrams, respectively, which are an aggregation of classes. We discovered a link from an use case describing the recording of object movements on the airport to a class diagram representing the entity model of trackable objects and their location data. This link allows to update the entity model in case the use case is changed or to trace from the use case to the classes in order to get a more detailed view on the system. While the use case contains only the short text “The user should have the possibility to record the movement of objects for short periods of his/her absence so that he/she can plot them at a later time. (Not available for normal employees).”, the class diagram contains many classes (over 20), most of them only containing a class name such as Car, which is a subtype of trackable objects. For a content-based approach, it is likely to be difficult to relate the use case with the diagram, since content is sparse in both entities and the only relationship between them is that all the objects in the diagram are movable.

We ran our recovery with different parameters to optimize the results. The best result from a run with a session timeout of 20 minutes, a transitivity-depth of 10, support and confidence thresholds of 1.5E-3% and 30%, respectively, is shown in table 1. The table lists source and target element types and the number of links in forward and backward direction of these link candidates as well as a total count for each link type. Although we consider the links to be bidirectional, it is interesting to see in which direction they were discovered since this also implies the most frequent direction of usage. Links with only few instances were aggregated into classes such as System Model Element and Project Model Element (see 3.1. Unexpectedly, many of the traceability links originate from or point to elements of the Project Model. We would have expected more links within the system model, especially from requirements to use cases. By examining the input data, we figured out that there are only few navigation instances from requirements to use cases and vice versa. The requirement and use case model are not consistent with each other. There are requirements that are not required for any use case; in other words, use cases are missing. By contrast, there are use cases that are not reflected in the requirements. We discovered the reason for this mismatch by means of the conducted interviews. While some of the teams worked mostly with use cases, other teams relied on requirements as primary representation for the customer’s requirements. Therefore there has been hardly any navigation from requirements to use cases. On the other hand, it

LinkSourceType	LinkTargetType	Fwd	Bwd	Total
Action Item	Functional Requirement	38	17	55
Action Item	System Model Element	6	0	6
Action Item	Meeting	4	0	4
Action Item	Rationale	21	6	27
Action Item	Workpackage	15	0	15
Class	Action Item	6	0	6
Class	Subsystem	17	0	17
Class	Scenario	1	0	1
Class	UML Diagram	9	2	11
Design Goal	Scenario	4	0	4
Meeting	Action Item	5	0	5
Meeting	Comment	4	0	4
Meeting	System Model Artifact	8	1	9
Rationale	Meeting	17	7	24
Rationale	Rationale	20	0	20
Rationale	System Model Element	5	1	6
UML Diagram	Project Model Element	3	0	3
UML Diagram	System Model Element	12	0	12
Use Case	Class	4	0	4
Use Case	UML Diagram	4	0	4
<b>Total</b>		203	34	237

**Table 1.** Classification of the discovered traceability links. Fwd/Bwd denotes the number of traceability links in forward/backward direction.

was a prerequisite from project management to report on the completion of the system in terms of open tasks. This is probably the reason for many navigation instances from Project Model to System Model, resulting in many recovered links in this segment. This is also reflected in the resulting model, in which almost 50% of the action items are related to a requirement, but less than 10% of the requirements are linked to use cases. In summary, 22% of the recovered links belong to the system model, 36% point from the system model to the project model, and 42% belong to the project model. From these results, we conclude that our approach is very sensitive to the actual usage of the model. We can only derive traceability links which are frequently used. This may sound like a disadvantage, but indeed it is an advantage: we only discover traceability links, which would have been used frequently.

To assess the quality of the recovered links, we interviewed several project participants. The test set was randomly selected from all candidate links. From a total of 43 links, 5 links were rejected. For 12 links the probands could not tell whether these actually represent traceability links. The remaining 26 links were accepted. In the overall experiment, the probands considered more than 60% of the candidates as meaningful traceability links, even when discarding the links the probands were not sure about. The problem with evaluating the overall quality of our approach is that by design the overall number of traceability links is unknown and very hard to estimate. Nevertheless, any re-

covered traceability link represents value added for the respective development project, unless the effort to review the recommended candidate links is overwhelming. Therefore the number of recommended candidate links can be controlled via the mining parameters support and confidence. Since it is likely that a higher confidence value of an association rule implies a higher applicability of the corresponding traceability link, we can cut down the result size by increasing the minimal confidence. Furthermore, a higher support value implies a higher applicability of the discovered rule within the space of all model elements.

## 6 Conclusion and Future Work

In this paper, we proposed a novel approach called ReARM for discovering traceability links based on association rule mining and operation-based change tracking. Our approach operates on a broad set of artifacts as part of a unified model including UML models and project management models such as schedules. In contrast to content-based approaches, we do not only consider the model at a certain point in time, but we take its history into account, and thereby also the activity of the developers working on the model (activity-centric). This results in two main advantages: First, content-based approaches such as similarity search using latent semantic indexing (LSI) do not address traceability as an activity that is carried out by developers in a software project, but only consider their static work products such as requirement documents or use case descriptions. Recovered links are not rated by frequency of use and might thus be of limited interest for the actual activities performed in a specific software development project. Second, since we do not rely on content, we can discover links even in situations where content is sparse or the involved artifacts are not related in content, but nevertheless require a traceability link. These situations often occur when a model is still under development and not yet complete, correct, nor consistent. Our approach can be used in addition to existing content-based approaches, completing their results with links that are not related in content but according to their change history.

We applied the ReARM approach in an industrial case study to show its feasibility and evaluate its results. We conclude that our approach represents a useful alternative to content-based approaches, especially in situations where these approaches fail: Traceability links which do result not from similarity in content and links among artifacts with sparse content. The ReARM approach is not limited to traceability link recovery and also shows potential for other applications. In future work, we will apply our technique to change impact analysis scenarios. Cost and time metrics based on the result of ReARM can be used to quantify the impact of a change, which in turn may support a go/no-go

decision of the project's change committee.

## References

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on management of data, Washington, D.C.*, pages 207–216, 1993.
- [2] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.
- [3] C. Böhm. Script 'knowledge discovery in databases'. Technical report, 2003.
- [4] J. Cleland-Huang, C. K. Chang, and M. Christensen. Event-Based traceability for managing evolutionary change. *IEEE Trans. Softw. Eng.*, 29(9):796–810, 2003.
- [5] J. Cleland-Huang, R. Settini, C. Duan, and X. Zou. Utilizing supporting evidence to improve dynamic requirements traceability. In *Requirements Engineering, 2005. Proceedings. 13th IEEE International Conference on*, pages 135–144, 2005.
- [6] J. David. Recommending software artifacts from repository transactions. In *The Twenty First International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems (IEA/AIE 2008), LNAI 5027*, pages 189–198. Springer-Verlag Berlin Heidelberg, 2008.
- [7] R. Dörmges and K. Pohl. Adapting traceability environments to project-specific needs. *Commun. ACM*, 41(12):54–62, 1998.
- [8] A. Dutoit, J. Helming, M. Koegel, and T. Wolf. Sysiphus project home. <http://sysiphus.org>, 2009.
- [9] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, page 163. IEEE Computer Society, 2002.
- [10] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 190–198, 1998.
- [11] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*, pages 13–23, 2003.
- [12] O. Gotel and C. Finkelstein. An analysis of the requirements traceability problem. In *Requirements Engineering, 1994., Proceedings of the First International Conference on*, pages 94–101, 1994.
- [13] J. Hayes, A. Dekhtyar, and S. Sundaram. Advancing candidate link generation for requirements tracing: the study of methods. *Software Engineering, IEEE Transactions on*, 32(1):4–19, 2006.
- [14] J. Helming, M. Koegel, and H. Naughton. PAUSE: a project analyzer for a unified software engineering environment. In *Workshop proceedings of ICGSE08*, 2008.
- [15] M. Koegel. Towards software configuration management for unified models. In *ICSE '08, CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 19–24, New York, NY, USA, 2008. Technische Universität München, ACM.
- [16] D. Leffingwell. Calculating the return on investment from more effective requirements management. Technical report, 1997.
- [17] M. Lormans and A. van Deursen. Can LSI help reconstructing requirements traceability in design and test? In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 10 pp.–56, 2006.
- [18] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4):13, 2007.
- [19] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, Toronto, Ontario, Canada, 2001. IEEE Computer Society.
- [20] J. I. Maletic, E. V. Munson, A. Marcus, and T. N. Nguyen. Using a hypertext model for traceability link conformance analysis. Technical report, 2003.
- [21] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27(4):364–380, 2001.
- [22] J. Richardson and J. Green. Automating traceability for generated software artifacts. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 24–33. IEEE Computer Society, 2004.
- [23] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 432–444. Morgan Kaufmann Publishers Inc., 1995.
- [24] R. Settini, J. Cleland-Huang, O. B. Khadra, J. Mody, W. Lukasik, and C. DePalma. Supporting software evolution through dynamically retrieving traces to UML artifacts. In *Software Evolution, 2004. Proceedings. 7th International Workshop on Principles of*, pages 49–54, 2004.
- [25] A. von Knethen and M. Grund. QuaTrace: a tool environment for (Semi-) automatic impact analysis based on traces. In *Proceedings of the International Conference on Software Maintenance*, page 246. IEEE Computer Society, 2003.
- [26] T. Wolf. *Rationale-based Unified Software Engineering Model*. Dissertation, Technische Universität München, July 2007.
- [27] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *Software Engineering, IEEE Transactions on*, 30(9):574–586, 2004.
- [28] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *Software Engineering, IEEE Transactions on*, 31(6):429–445, 2005.
- [29] A. Zisman, G. Spanoudakis, E. Perez-Minana, and P. Krause. Tracing software requirements artifacts. In *Proceedings of the 2003 International Conference on Software Engineering Research and Practice, SERP*, volume 3, 2003.