A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis

Thazin Win Win Aung, Huan Huo, Yulei Sui
University of Technology Sydney, Australia
thazinwinwin.aung@student.uts.edu.au,{huan.huo,yulei.sui}@uts.edu.au

ABSTRACT

In large-scale software development projects, change impact analysis (CIA) plays an important role in controlling software design evolution. Identifying and accessing the effects of software changes using traceability links between various software artifacts is a common practice during the software development cycle. Recently, research in automated traceability-link recovery has received broad attention in the software maintenance community to reduce the manual maintenance cost of trace links by developers. In this study, we conducted a systematic literature review related to automatic traceability link recovery approaches with a focus on CIA. We identified 33 relevant studies and investigated the following aspects of CIA: traceability approaches, CIA sets, degrees of evaluation, trace direction and methods for recovering traceability link between artifacts of different types. Our review indicated that few traceability studies focused on designing and testing impact analysis sets, presumably due to the scarcity of datasets. Based on the findings, we urge further industrial case studies. Finally, we suggest developing traceability tools to support fully automatic traceability approaches, such as machine learning and deep learning.

KEYWORDS

traceability, change impact analysis, natural language processing

ACM Reference Format:
Thazin Win Win Aung, Huan Huo, Yulei Sui. 2020. A Literature Review of Automatic Traceability Links Recovery for Software Change Impact Analysis. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389251

1 INTRODUCTION

Software change impact analysis (CIA) helps control software design evolution in the maintenance of continuous software development. Bohner defined CIA as "the assessment of the effect of changes – providing techniques to address the problem by identifying the likely ripple-effect of software changes and using this information to re-engineer the software system design" [8]. During software maintenance, a change can not only impact source code but also cause a ripple effect upon other artifacts (i.e., requirements, design and test cases) [15]. Gotel defined requirements traceability as "the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent development and use, and through periods of on-going refinement and iteration in any of these phrases)"[22]. Therefore, impact analysis can use traceability links to understand relationships and dependencies between various software artifacts.

In Bohner's impact analysis process model, identifying a set of current impacts is the first phase of the process model. In their model, there is a total of four impacts sets (i.e., requirement impacts, design specification impacts, program impacts and test impacts) that need to be identified in order to provide the complete sets of changes [8]. Traceability links are also needed for these impact sets. However, one of the main challenges of recovering traceability between software artifacts of different types is a knowledge gap problem [15] between artifacts. Knowledge in this context refers to syntax and semantics of the artifacts [15]. For instance, source code is written in the programming language, whereas requirements are documented in natural language; thus, dependencies between these two artifacts cannot simply be recovered by parsing the knowledge of the artifacts. In addition, software engineers need to verify the traceability links between software artifacts recursively until they identify all the impacted artifacts [9]. It is time-consuming and error prone. Therefore, a large and growing body of literature has investigated the automatic traceability links recovery approach to support impact analysis process.

Several studies have applied information retrieval (IR) approaches [4, 6, 19, 25, 36] to the problem of establishing traceability links between software artifacts of different types. Recently, deep learning (DL) approaches have been applied in studies to leverage the accuracy of recovering traceability links recovery between the requirements and the design artifacts [24, 32]. The rationale behind these approaches is that software documentation (i.e., requirements, design, program comments, commit logs and test cases) are expressed in natural language, and the high textual similarity between two artifacts has high potential to share the same context. Based on this assumption, traceability links can be recovered between software artifacts of different types. Also, machine-learning (ML) approaches focus on optimising the time and effort of verifying impact candidate links. Several studies have also presented the effect of transitive tracing on traceability-link recovery [10, 21, 35], which recovers traceability links between two artifacts by joining them to a third artifacts. This approach is useful when the source and target artifacts share little textual similarity, as a third artifact can be a transitive artifact to connect the other two. Automating traceability-link recovery simplifies the software engineer's task in identifying the complete impact change sets.

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389251

The aim of this study is to systematically review automatic traceability links recovery approaches in a software CIA context to identify research gaps and report evidence to direct future research. The term automatic in this paper implies both semi-automatic and fully automatic approaches. We classified studies as semi-automatic if their processes required user input to manually filter out false positive trace links (e.g., IR-based approaches) and as automatic if they required no user input to verify trace links (e.g., ML approaches). We clustered publications based on traceability approaches, change impact sets, trace direction and degree of evaluation. We followed the systematic literature review guidelines provided by Kitchenham and Charters[28] and reviewed the primary studies published during 2012–2019. Our SLR paper aims to answer the following research questions:

- RQ 1. What approaches have been adopted to recover traceability links between artifacts to support CIA?
- RQ 2. Which change impact sets have been covered?
- RQ 3. How have studies adopted transitive tracing approaches to recover traceability links between artifacts?

The paper is structured as follows: Section 2 provides a background and overview of existing reviews relevant to automatic traceability link recovery. Section 3 presents our research objectives and RQs and describes our review method, including search strategy, selection process, quality assessment, data extraction and data analysis. Section 4 summarises the key findings of our study. Lastly, section 5 discusses the meaning of findings and the study's limitations.

2 BACKGROUND

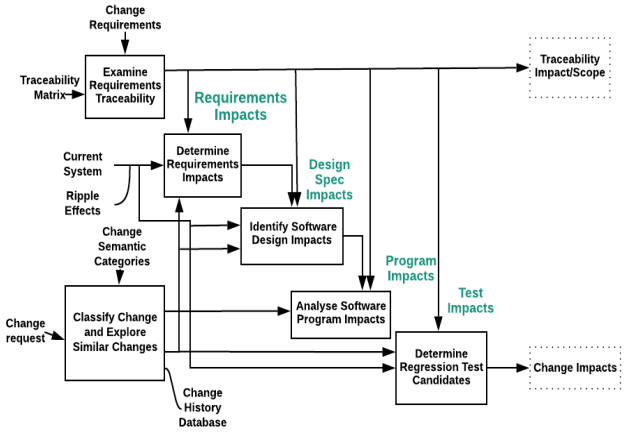

Figure 1: Identify Software Change Impacts [8]

Bohner [8] earliest work of change impact process model represents the key activities in identifying software impacts. Figure 1 presents the Bohner's change impact analysis process model. The traceability activity is broken down into five sub-activities as follows.

- **Classify change and explore similar changes** - This activity assesses the change history data to compare the current change request with similar software changes from the system's change history database.
- **Determine requirements impacts** - This activity determines the requirements impacts by associating new requirements with current system requirements history information.
- **Identify software design impacts** - This activity examines the current system architecture and program design information passed from *classify change and explore similar changes* and identifies design impacts based on the information guided *examining requirements traceability* and *determine requirements impacts* activities.
- **Analyse software program impacts** - This activity uses various program analysis techniques such as program slicing, data flow analysis, control flow analysis and dependency analysis to determine program impact sets.
- **Determine regression test candidates** - This activity uses the current system test information from *classify change and explore similar changes* activity, guided by information from program impacts activity and requirements traceability activity to determine test impacts.

Bohner's process model is specifically designed for the traditional software development approaches which rigidly produced the requirements documents. Therefore, most existing traceability approaches are focused on establishing the links between requirements and their downstream artifacts (i.e., design, test case and code)[2, 4, 6, 36, 40]. However, modern software development projects based on agile methodologies omit comprehensive documentation. In [13, 18], the authors described the challenges of using traditional traceability approaches in agile software projects. Popular agile approaches such as eXtreme Programming [5] and Scrum [41], adopt test-driven development, which captures the initial requirements as user stories in brief and transforms them into the scripted, customer-accepted tests before writing the code. In [13], Cleland-Huang proposed the traceability information model (TIM), which implicitly establishes traceability links from customer-accepted test cases to source code to support CIA in agile software projects. As our goal is to review automatic traceability approaches, a thorough discussion of agile-specific traceability approaches is outside the scope of this paper, although we highlight the need for test case–driven automatic traceability approaches to support agile software projects in future work.

2.1 Traceability Challenges

As mentioned in the previous section, recovering and maintaining traceability information throughout the software development is important for the CIA process. Based on Bohner's impact analysis model and Cleland-Huang's traceability model, it is required to establish traceability links between heterogeneous artifacts (e.g., requirements, design, source code and test case) [8, 13]. However, one of the main challenges of recovering traceability links between software artifacts of different types is a knowledge gap problem. Knowledge in this context refers to syntax and semantics of the artifacts [15]. There is a high level of knowledge gap between

software documentation and source code. The formal one is usually expressed in a natural language, whereas the latter follows program syntax and language. Recovering knowledge-based traceability links between these heterogeneous artifacts required data normalisation as well as human experts verification [6, 19, 25].

In recent decades, IR has been widely adopted as a core technology to address the problem of recovering knowledge-based traceability links between artifacts of different types [4, 6, 19, 25, 36]. This approach establishes traceability relationships on the assumption that, if two artifacts share high textual similarity, they are likely related. However, IR-based approaches require human experts to verify candidate trace links manually, as it is time-consuming and error-prone. Recently, several studies have proposed training ML classification models to verify the validity of trace links generated with IR approaches [1, 16, 17, 32, 39].

Indeed, several authors have reported the benefits of DL approaches over IR-based ones. DL approaches generally can learn unstructured data of any format, as the networks can be trained to understand the domain knowledge of the system [24, 32], such as understanding the correlations between requirements and design documents [23, 47]. To utilise the right traceability link–recovery approach, software engineers must understand which approaches are suitable for which change impact sets and, how trace links can be recovered and the availability of support tools.

2.2 Related Work

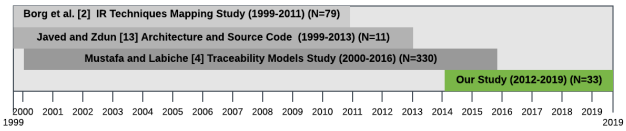

Figure 2: Overview of traceability literature review timeline

Figure 2 presents an overview of the timeline, the domain of automatic traceability link–recovery approaches and the total number of primary studies in existing literature reviews compared to the present paper. In Borg et al.[11], the authors presented a systematic mapping study of IR-based traceability link–recovery approaches and enhancement strategies covering studies published from 1999–2011. They highlighted a lack of empirical evidence of any IR–based model consistently outperforming another. Javed and Zdun[26]] reviewed studies on traceability published between 1999 and 2013, capturing the correlations between software architecture and source code. They highlighted that semi–automatic traceability approaches appeared the most appropriate ways to create trace links between software architecture and source code. Recently, Mustafa and Labiche[34] reviewed studies published between 2000 and 2016 on traceability models. However, none of these reviews discussed traceability link–recovery approaches using CIA, and no recent empirical studies have examined the current state of fully automatic traceability link–recovery approaches. As mentioned earlier, we reviewed studies that used both semi-automatic and fully automatic approaches to review the recent developments of both. Just as Borg et al.[11] and Javed and Zdun[26] reviewed studies of semi–automated approaches from 1999–2013, we reviewed studies published between 2012 and 2019 to study automatic traceability link–recovery approaches using software CIA to report empirical evidence and identify research gaps.

3 REVIEW METHOD

We followed SLR guidelines [28] and developed a protocol to plan, execute and report our results. We started our review process in mid-2019 and ended in late 2019. All authors of this study were involved in the review process. The following sections outline the processes included in our planning phases.

3.1 Objectives and Research Questions

Our goal was to gather the state of the literature on automatic traceability-link recovery under the context of CIA, so we strove to answer three complementary RQs specified by the following criteria:

RQ1. What approaches have been adopted to recover traceability links between artifacts to support CIA?
We investigated the approaches taken most frequently in traceability-link recovery studies as well as trace direction and degree of evaluation. We also studied whether the studies introduced any supporting tools.

RQ 2. Which change impact sets have been covered?
Based on Bohner's [8] impact analysis model, we analysed four change impact sets (i.e., requirements, design, program and test) and reported which traceability studies covered them.

RQ 3. How have studies adopted transitive tracing approaches to recover traceability links between artifacts?
We investigated the purpose of transitive tracing in this regard.

3.2 Protocol Development

In our SLR-planning phase, we initially searched for other SLRs with similar scopes. In our preliminary search, we found a few relevant studies that fit our research objectives (see Section 2.2). Accompanied by the already-identified studies, we used these SLRs as the basis for our own RQs and to develop our iterative review protocol. The protocol document included SLR RQs, search strategy, study selection criteria, quality assessment, data extraction strategy, and data synthesis and analysis guidelines, which are mentioned briefly in the following sections.

3.3 Search Strategy and Data Sources

Following the research objectives and RQs, we selected four important terms for searching the literature: (1) "traceability," (2) "recovery," (3) "software artifacts" and (4) "change impact analysis". We then selected a range of online databases, ran simple searches in the titles, keywords and abstracts of the publications, and reviewed the coverage. While running the searches, we created the search strings and modified them for various online databases.

ON ABSTRACT: (Abstract: trace*) AND (Abstract: recover* OR Abstract: maintain OR Abstract: link OR Abstract: establish) AND (Abstract: requirement OR Abstract: specification OR Abstract: architecture OR Abstract: design OR Abstract: code OR Abstract: implementation OR Abstract: test OR Abstract: bug) AND (Abstract:

change OR Abstract: impact OR Abstract: analysis OR Abstract: system comprehension)

3.4 Study Selection

We are interested in collecting peer-reviewed studies, published between 2012-2019, focused on the automatic traceability links recovery used for CIA. In a first step, we defined the selection criteria below.

- The publications are written in English
- Research explicitly mention they are targeting trace recovery relating to software maintenance and change impact analysis
- Studies contain empirical results (e.g., case study, experiments and surveys)

Figure 3 shows a summary of search and selection process.

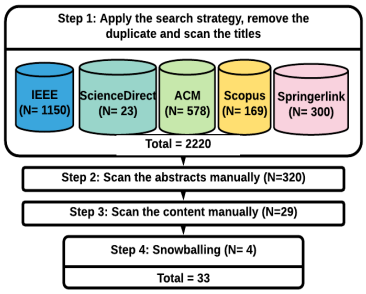

Figure 3: Search and selection process

- **Step 1:** We applied our search strings in the following five databases: (1) ACM Digital Library, (2) IEEE xplore, (3) Science Direct, (4) SpringerLink and (5) Scopus. Next, we discarded the duplicate papers. Then, we scanned the title and excluded the irrelevant papers. The search results showed a high number of documents (2220 findings).
- **Step 2:** To extract the results, the first author manually scanned the abstracts based on selection criteria. Then, the second and third authors checked a sample of papers randomly. The differences were resolved in discussion between authors. We included 320 studies in this step.
- **Step 3:** At the beginning, the first author scanned the content of the papers based on the selection criteria and marked the records as relevant/irrelevant studies. The second and third authors reviewed the findings and selected the relevant studies separately. We found 29 studies in this step.
- **Step 4:** Finally, the first author performed a forward snowballing on our included studies that have high citations. The another authors reviewed the new findings. We found four additional papers in this step. In total, we identified 33 relevant studies.

3.5 Quality Assessment

We performed a quality assessment in two stages as follows:

- **Assessment of research design:** To assess the quality of studies, we reviewed the research design mentioned in the papers. This includes assessing the details of research objectives, design and evaluation. Therefore, we filtered out the studies that have poor research methods and evaluation as well as research objectives not related to automatic traceability links recovery. We used the quality assessment checklist from [28] [1]).
- **Assessment of publication source and impact:** We checked the professional computer science CORE[2] journal/conference ranking site to evaluate the quality of the publications' sources. The CORE executive committee periodically update the ranking for addition or re-ranking of conferences by surveying academics from worldwide as the classification is internationally acknowledged ranking.

3.6 Data Extraction and Analysis

To answer the RQs, we extracted the following demographic data for review: title, authors, type of outlet (journal or conference), name of outlet, publication year, type of trace artifacts, trace direction, trace recovery technique, quality of evaluation (e.g., research design), CORE[2] ranking and CIA coverage.

4 RESULTS

We have included 33 relevant studies in this review. Concerning the publication channel, the studies were published in conference proceedings, workshops and scientific journals. In comparison, 19 papers (58%) of the included studies where published in conference proceedings, 11 papers (33%) appeared in journals and 3 papers (9%) belonged to workshops. We identified that 30 papers of the included studies were published in high ranking conferences and journals. Only three papers cannot verify ranking, but these papers were published in the traceability specific journal and workshop. Please see the primary classification here[3].

We included 33 relevant studies in this review. In total, 19 (58%) were published in conference proceedings, 11 (33%) appeared in journals and 3 (9%) belonged to workshops. We identified 30 that were published in high-ranking conferences and journals. We could not verify the ranking of only three studies, but they were published in traceability-specific journals and workshops (please see our primary classification[3]).

4.1 Traceability Links Recovery Approaches Between Software Artifacts

Several approaches have been proposed to recover knowledge-based traceability links between software artifacts of different types to assist in the impact analysis process. Please see our primary classification [3]. Figure 4 illustrates the publications trend grouped by traceability approaches and artifacts. We found a total of four approaches used in the studies (i.e., IR, heuristic, DL and ML). The following sections review the purposes of these approaches in the traceability context.

4.1.1 Information retrieval–based approaches.

In the early years of the field, several studies applied IR approaches

[1] https://doi.org/10.6084/m9.figshare.11955459
[2] http://www.core.edu.au
[3] https://doi.org/10.6084/m9.figshare.11955900

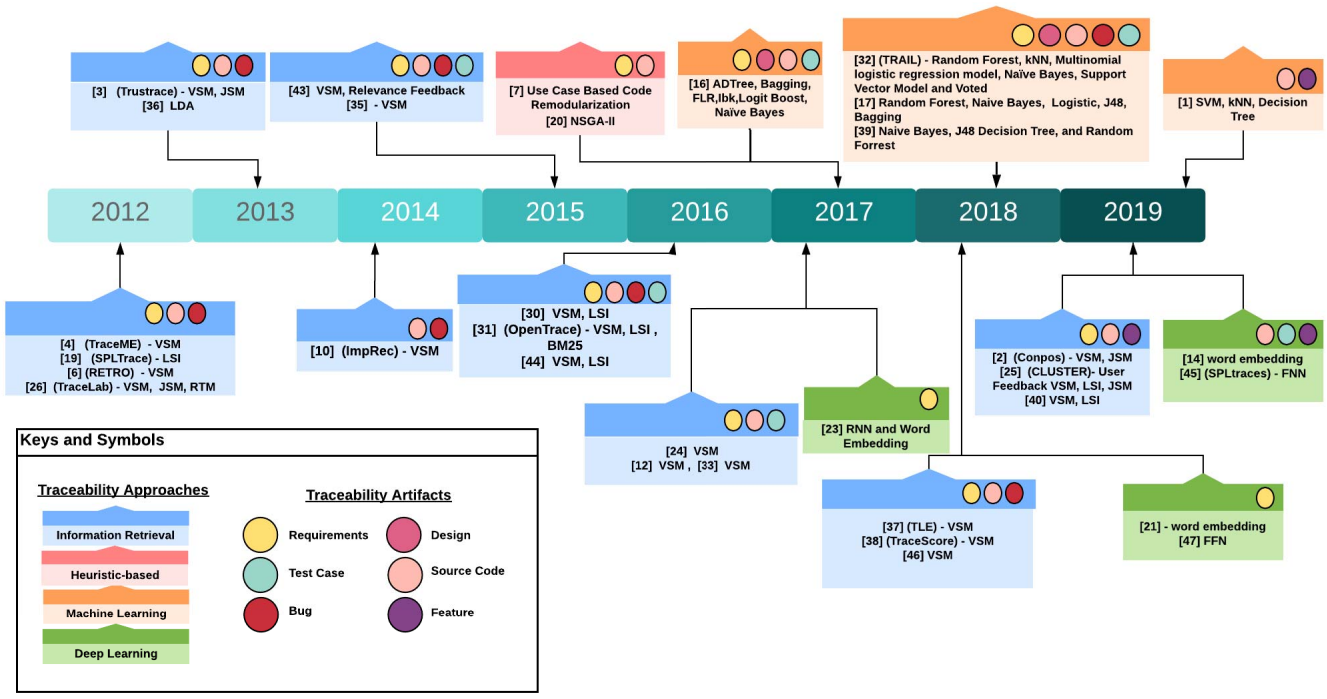

Figure 4: Traceability link recovery publication trends (A rectangular box is colored according to traceability approaches. A circle symbol insides the top-right corner of the rectangle box represents the trace artifacts applied in the studies. Texts inside the rectangular box present the traceability approaches or tool names and techniques.)

to tackle the challenges of recovering knowledge-based traceability links between various software artifacts (i.e., requirements, test cases, source code, bug reports and features). In total, 21 of the 33 studies used IR approaches to identify various change impact sets because most of the software artifacts were written in a natural language, as the links between artifacts of different types can be recovered by computing the textual similarities between them (a high textual similarity between two artifacts is assumed to relate them to each other in these approaches). We identified vector space models (VSMs) [4, 6, 19, 36], latent semantic indexing (LSI) [19, 30, 44], Jensen and Shannon models (JSMs) [3, 25] and latent Dirichlet allocation (LDA) [36] as the most commonly applied techniques in the studies, likely due to their easy implementation and set-up.

Ben Charrada et al.[6] presented an IR-based approach to identify outdated requirements by monitoring source-code changes. All too often, software engineers directly modify source code without updating the corresponding requirements. As a consequence, traceability links between these two artifacts become obsoleted and cannot be used in impact analysis effectively. To mitigate this problem, the authors proposed extracting trace query terms from recent source-code changes (e.g., the addition of a new method, class or package or the deletion of an existing method, class or package) to establish links with the corresponding requirements. Similarly, Gethers et al.[19] illustrated the use of LSI to recover traceability links between bug reports and source code to estimate impact sets. In contrast, their approach extracted source-code query terms based on the commit change-sets co-occurrence concept (e.g., method A and method B are committed to three commit transactions together, which are considered "co-occurrence" methods). Based on this concept, the authors established the most relevant trace links between bug reports and source code. Ali et al.[3] enhanced identification of outdated requirements by monitoring source-code changes in bug-fix histories to identify the impacts on original requirements. Similarly, Unterkalmsteiner et al.[44] experimented with VSM and LSI techniques to recover traceability links between source code and test cases to identify test case impact sets.

Differently, Panichella et al.[36] proposed a source code class–based, topic-modelling approach (i.e., LDA) to establish traceability links between source code and requirements. The rationale behind this approach is that a class is an abstraction of a domain/solution object and a use case is homogeneous and related to one specific topic. Later that year, Borg, Rath et al.[10, 38] presented the idea of recovering traceability links between a new issue report and previous issue reports in an issue repository to identify a set of potentially impacted artifacts [10, 38]. The study is based on the assumption that previous issue reports have more textual similarity relations with current implementation artifacts than a new issue report. In Tsuchiya et al.[43], the authors proposed an approach to combine two similarity relevance scores from two sets of traceability links (i.e., one between requirements and source code, another one between requirements and commit messages) to improve the accuracy when establishing links between requirements and source

code. Similar to Borg[10], Nishikawa et al.[35] presented an approach called connecting link method (CLM) to recover transitive traceability links between two artifacts using a third artifact to overcome the problem of source and target artifact with no textual similarity between them. The approach is based on the assumption that if a requirement 1 is implemented in test case 1 and test case 1 is related to source code 1, then source code 1 will include the implementation of a requirement 1.

Similarly, Mills and Haiduc[33] studied the impact of trace-retrieval direction on the accuracy of trace-link recovery between requirements and code classes. They used a standard VSM and established the trace links between artifacts bidirectionally. Interestingly, their results indicated a high correlation between the accuracy of an IR-based traceability-recovery approach and trace direction. Recently, Rahimi and Cleland-Huang[37] presented an approach called trace link evolver (TLE) to detect obsolete trace links between requirements and source code by identifying changes between two consecutive versions of source code. They used a source-code refactoring tool to detect a wide range of source-code change scenarios (e.g., add, remove, move, rename and merge) at both the method and class levels. Once the code changes were identified, they used a standard VSM to recover update-to-date links between the requirements and the source code. Their findings indicated that trace links generated with TLE have higher precision than those generated with a standalone VSM technique.

In [30], Mahmoud and Williams presented a taxonomy-based traceability-link recovery approach to establish trace links between non-functional requirements and source code. They manually defined non-functional requirement query terms and maintained them in a custom taxonomy database. Their IR engines, which they based on VSM and LSI models, used these taxonomy terms to establish trace links between non-functional requirements and source code. Likewise, Guo et al.[24] proposed a taxonomy-based traceability-link recovery approach to establish the traceability links between Health Insurance Privacy and Portability Act (HIPAA) regulations and system requirements documents. In [46], the authors similarly created a requirement-based taxonomy approach to recover trace links between requirements and source code.

Unlike [3, 19], Chaparro et al.[12] proposed an approach to extracting observed behaviours terms from bug reports to leverage an IR-based approach. Observed behaviours are texts that describe the misbehaviour of a system (e.g., "the menu doesn't open when I click the button"). They manually extracted observed behaviour terms from bug reports and used a standard VSM model to recover traceability links between bug reports and source code. Similarly, Ali et al. [2] built a parts-of-speech (POS) tagging method with a constraint-based pruning approach to improve accuracy. Their approach first extracted all POS categories (i.e., nouns, verbs, adjectives, adverbs and pronouns) from requirements documents and then computed textual similarity between source-code identifiers (e.g., class and method) using a VSM and JSM.

Recently, [25] proposed an approach called closeness-and-user-feedback-based traceability recovery"(CLUSTER), which establishes trace links between requirements and source code based on code dependencies among classes to improve the accuracy between the links. Their approach calculates code dependencies based on degrees of direct interaction (e.g., method calls, inheritance and class usage) and indirect interaction (e.g., reading or writing the same data). Recently, Seiler et al.[40] proposed a feature-tagging approach to recover traceability links between requirements and source code. A feature is a short textual description of functionality that presents business value, whereas a feature tag concisely summarises the feature. Their approach recommends labelling requirements and source code with corresponding tags during development and using those tags.

4.1.2 Heuristic-Based Approaches.

In previous IR-based approaches, a human expert needed to repeatedly and manually verify candidate link lists, which is time-consuming and error-prone. To reduce the time and effort of the verification process, Berta et al.[7] presented a traceability-recovery approach based on multiple search criteria to establish trace links between requirements and source code. They experimented with a non-dominated sorting algorithm (NSGA–II) with three weighting criteria (i.e., similarity scores, frequency of change and recency of change) and used cosine similarity to calculate similarity scores between requirements and the source code. They extracted the metadata for the remaining two criteria—source-code frequency of change and recency of change —from source-code version history. This approach established trace links based not only on semantic similarity between software artifacts but also on change history, optimising the accuracy of candidate link lists. All their experimental results harmonically achieved high-precision results.

In contrast, Berta et al.[7] enhanced the IR-based approach with a novel method called re-modularization by transforming source-code syntax into natural language sentences. These sentences are then used to establish trace links between the source code and the use cases. Converting source-code syntax to natural language reduced the knowledge gap between artifacts when calculating the similarities between them.

4.1.3 Machine Learning-Based Approaches.

We found that most primary studies adopted ML approaches to automatically verify candidate link lists generated from IR approaches [16, 17, 32]. In [1], the authors used ML classifiers to recommend relevance features to comments in the source code.

Falessi et al.[16] proposed an approach called estimation of the number of remaining links (ENRL) to detect the remaining number of positive trace links from ranked lists generated with IR approaches. They trained seven ML classifiers (i.e., ADTree, Bagging, Fuzzy Lattice Reasoning, IBk, Naïve Bayes, LogitBoost and ZeroR) with a set of classified golden standard trace links to identify positive and negative links from the ranked lists. Their results indicated that the ZeroR classifier produced the lowest-accuracy results with a mean relative error (MRE) of 1. Similarly, Mills et al. [32] presented a framework called traceability links classifier (TRAIL) to automatically verify the validity of ranked trace links generated with an IR model. They used three features to validate the trace links: cosine similarity, query quality metrics and document statistics. They experimented with TRAIL with six ML classifiers: Random Forest, k-Nearest Neighbours (kNN), Multinomial Logistic Regression Model, Naïve Bayes, Support Vector Model (SVM), and Voted. Their findings indicated that the Random Forest classifier outperformed the other five. Also, Falessi et al.[46] trained Random Forest, Naïve Bayes, Logistic, J48 and Bagging models to verify the validity of

trace links between requirements and source code, but they did not clearly report which ML classifier performed best. Likewise, Rath et al.[39] proposed an approach by training ML classification models to predict potential trace links between issue reports and source code. They used Naïve Bayes, J48, Decision Tree (DT), and Random Forest classification models to validate the possible trace links, finding that Random Forest outperformed the other two.

In contrast, Abukwaik et al.[1] proposed a feature-annotation recommendation approach, suggesting that developers add annotation features in newly added code before check-in into a version-control system. They used three ML classification models (i.e., SVM, kNN and DT) to predict source-code locations where feature annotations were missing. The approach used previous change-sets history to formulate lists of possibly related features for new changes. Their results indicated that the kNN classifier produced the highest accuracy after training with 60 commit change-sets history.

4.1.4 Deep Learning-Based Approaches.

Recently, DL techniques have become popular in the traceability context to address the knowledge-gap problem. We found four studies that used recurrent neural networks (RNNs) [23], feedforward neural networks (FNNs) [47] and word-embedding [14] to recover traceability links between software artifacts of different types (i.e., requirements, source code, test cases and features). In [14], the authors demonstrated that a word-embedding approach outperformed an LSI model when establishing trace links between test cases and source code.

Guo et al.[23] combined the word-embedding and the RNN approaches to eliminate knowledge gap in recovering traceability links between requirements and design documents. The approach can be divided into two layers: a word-embedding mapping layer and a semantic-relation evaluation layer. In the first layer, it converted the collection of requirements documents to a word-embedding vector format and identified semantic relations between terms using RNN. Conversely, the same process ran for design documents. The outcomes of the embedding layer were then passed to the evaluation layer to recover the links between them. In the evaluation layer, it first calculated the vector distances and directions between pairs of semantic vectors and then passed the resulting vectors to the sigmoid and softmax (e.g., 1- valid, 0- invalid) functions to calculate the relations between them.

Similarly, Wang et al.[47] used the word-embedding and the FNN approaches to address the polysemy issue in recovering trace links between requirements. Polysemy, in this context, refers to "the coexistence of multiple meanings for a term appearing in different requirements" [47]. In their approach, the authors extended the standard IR recovery approach with the term-pair ranking model and cluster-ranking model. The term-pair ranking model identified the lists of polysemy terms in a requirements collection using word-embedding and FNN. The cluster-ranking model then used these polysemy terms and updated the term-to-requirements matrix accordingly. They evaluated their approach with two baseline IR models (i.e., the VSM and LSI). Their results indicated that the accuracy of the results increased by eliminating the polysemy issue. Recently, Csuvik et al.[14] trained a word-embedding model with raw source code, an abstract syntax tree structure of source code and a test case to predict relevant test cases. The authors compared their word-embedding model with a standard LSI model and reported that word-embedding outperformed LSI when recommending the most similar class.

Table 1: Traceability tools

Tool	Features
TraceME [4]	An eclipse-plugin application to recover the traceability links between source code and requirements during software development. The tool supports a graph view to visualise the trace links between source code and requirements.
RETRO [6]	It is a web-based traceability recovery application to establish trace links between artifacts of different types (i.e., requirements, test case and source code).
TraceLab [27]	A Winforms traceability workbench application to experiment with traceability research projects. The framework provides a collection of customizable and configurable IR components to establish traceability links between various types of textual artifacts of a software project. The application is built on top of the General Architecture for Text Engineering (GATE) framework.
OpenTrace [31]	A Winforms experimental traceability workbench application for researchers to perform traceability research projects. The tool supports built-in as well as custom-built traceability components (i.e., data importers, pre-processors, IR- based trace algorithms, trace matrices, debugging utilities and result exporters) to establish links between different software artifacts. The application is built on top of the C#.Net framework.
SPLTrace [45]	An experimental traceability framework built on four IR models (i.e., Class Vector, Extended Boolean, Latent Semantic index, and BM25) and one deep-learning model (i.e., Feedforward Neural Network). The framework provides the components to recovery traceability links between project features and source code. The application developed using Python.

In terms of tool support, we identified five traceability tools (i.e., TraceM [4], RETRO [6], TraceLab [27], OpenTrace [31] and SPLTrace [45]). Table 1 presents the traceability tools proposed in the primary studies. Generally, TraceMe and SPLTrace can be categorised as special-purpose tools focused on identifying the impact set between requirements and source code. In contrast, RETRO can be considered a general-purpose tool suited to analysing the impacts between software artifacts of different types. Finally, TraceLab and OpenTrace can be grouped as workbench tools that support various IR-based trace-recovery components (e.g., data preparation, experiment execution and evaluation), as it is applicable to experimental analysis of various IR approaches.

4.2 Traceability Direction and Evaluation

In [33], the authors highlighted the impact of trace direction on the link-recovery process. We thus grouped the studies based on two traceability directions (i.e., unidirectional and bidirectional). Figure 5 shows the classification of the studies by trace direction.

In the unidirectional traceability group, we included the 31 (94%) out of the 33 studies that evaluated their approach to recovering trace links from one artifact to another in either direction (e.g., requirements to design, requirements to source code, etc.). Only two studies [33, 35] evaluated their approaches bidirectionally. To the best of our knowledge, the approaches focused on tracing between two natural-language artifacts (e.g., requirements, test cases and design) are possible to trace bidirectionally [23, 47]. However, approaches such as feature-based tracing [1, 24, 40] and code-to-requirement tracing [6, 14] require specific data tokenisation for

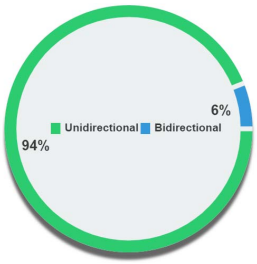

Figure 5: Distribution of publications by traceability direction

source code, as these approaches may not be applicable to trace bidirectionally. Due to the lack of evaluation in the studies, we cannot conclude which approaches are applicable to bidirectional tracing.

In terms of degree of evaluation, we reported the research methods and the type of datasets used in the studies. We used the quality-assessment checklist from [28][1] to assess their research methods.

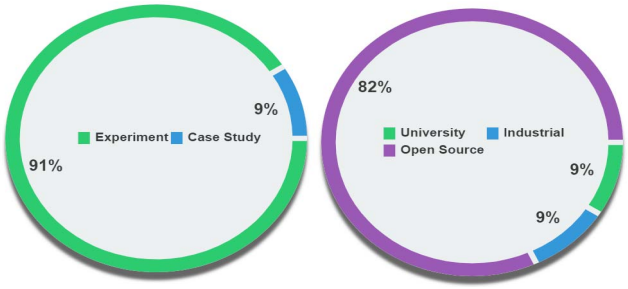

Figure 6: Distribution of publications by research methods (left) and datasets (right)

Figure 6 presents the research methods and datasets used in the primary studies. We found that 30 (91%) out of the 33 studies applied experimental research to study the effectiveness of their approaches. Only 3 (9%) of them [10, 43, 44] used case study methods. In [10], the researchers studied the impact of automatic traceability-based CIA approaches in two industrial domains (i.e., automation and telecommunication). Their findings indicated that their approach could identify 40% of the potential program change set effectively in real-world datasets. Similarly, [43] applied a standard VSM model in a Japanese software company to establish the trace links between requirements and source code.

Due to data confidentiality, the study anonymised the system details. In [44], the authors experimented with the two IR models (i.e., VSM and LSI) in an embedded system, which produces both hardware and software products. The study evaluated IR models by establishing links between the source code and test cases. Based on their findings, an IR-based approach could only identify 38% of the potential test cases in industry setting. We found that 27 (82%) out of the 33 studies used open-source projects to evaluate their approaches, which are presumably closed to industrial projects. Only three studies [4, 6, 35] used university projects to evaluate their traceability tools (RETRO and TraceMe).

4.3 Support Change Impact Set

This section classifies the primary studies based on Bohner's four impact sets (i.e., requirements, design, program and test) and presents the results in Table 2, the last row of which describes the acronyms used in the table.

Table 2: Distribution of publications by change impact sets

Impact Set	Artifacts Links	Studies
RIS	SC-R, RR-R, R-R, F-R	[6],[4], [3], [35], [24], [33], [16], [23],[27], [32], [46], [47], [40]
DIS	R-D	[35], [16],[27]
TIS	SC-TC, R-TC, D-TC	[35], [44], [16],[32],[27]
PIS	R-SC, Bug-SC, TC-SC, F-SC	[19],[36], [10], [43], [35], [30], [31], [7], [20], [12], [33], [16], [38], [21], [17], [46], [39],[37], [1], [2], [45], [40], [14], [27]
Acronyms: RIS - Requirements Impact Set, DIS - Design Impact Set, TIS - Test Impact Set, PIS - Program Impact Set, R- Requirements, RR - Regulatory Requirements, F - Features, D - Design, SC - Source Code, TC - Test Case		

4.3.1 *Requirements Impact Set.*

This section includes the studies that used requirements as target artifacts in their trace-recovery approaches to identify the impact on requirements levels. We identified four source artifacts (i.e., regulatory requirements [24, 46], low-level requirements [23, 47], source code [3, 4, 6, 16, 27, 32, 33, 35] and features [40]) used in the studies to assess the requirements impact scope. In [24, 46], the authors recovered links between regulatory requirements and requirements using a taxonomy-based approach to identify the impact on the requirements level. Similarly, [23, 47] used a DL-based approach to recover trace links between high-level and low-level requirements.

Similarly, in [3, 4, 6, 33, 35], the authors mined the class- and method-level changes of source code from commit histories and established the links with existing requirements to identify outdated requirements. In [16, 32], the authors trained ML models to predict trace links between source code and requirements. Recently, Seiler et al.[40] introduced a feature-tagging approach to maintain the links between requirements and other software artifacts. Interestingly, we could not find studies that used design and test cases as source artifacts, presumably due to scarcity of test datasets.

4.3.2 *Design Impact Set.*

In this section, we included studies that used design artifact as a target artifact in the primary studies. Interestingly, only three studies [16, 27, 35] focused on recovering links between requirements and design artifacts, possibly due to scarcity of datasets. These studies [16, 27, 35] evaluated their approaches with the same test datasets, due to the availability of golden standard answer sets. In [27, 35], the authors used standard IR models to recover the links between requirement and design artifacts. In [16], the study used the machine learning-based approach to verify the candidate links

between requirements and design artifacts, which are generated with the IR engine.

4.3.3 *Test Impact Set.*

This section includes studies that used tests as target artifacts. We identified two source artifacts (i.e., design [16, 27, 35] and source code [27, 32, 44]). In [16, 35], the authors evaluated using the same datasets to recover links between design and test case artifacts to identify test impact sets. In [44], the authors used two IR models (i.e., VSM and LSI) to identify test impact case sets by linking with source-code artifacts. Similarly, in [32], the authors used an ML approach to automatically verify the candidate links between source-code and test artifacts generated with an IR approach.

4.3.4 *Program Impact Set.*

This section includes studies that used source code as a target artifact. We identified four source artifacts (i.e., requirements [2, 7, 16, 17, 19, 20, 25, 27, 30, 33, 36, 37, 43, 46], bug reports [10, 12, 21, 31, 38, 39], test cases [14, 35] and features[1, 40, 45]) used in the studies to identify the program impact set. In [2, 19, 25, 30, 33, 36, 37, 43, 46], the authors applied an IR-based approach to establish trace links between requirement and source-code artifacts to identify program impact sets. To improve accuracy, the study used a heuristic-based approach in [7, 20]. In [16, 17], the authors trained ML models to validate trace links between requirements and source-code artifacts. Many authors employed IR-based approaches [10, 12, 21, 31, 38] and ML approaches[39] to recover trace links between bug reports and source code.

Likewise, one study [35] applied the standard VSM model to recover trace links between a test case and source code. In [14], the study used a word-embedding model to establish trace links between a test case and source code to identify program impact sets efficiently. Recently, some studies introduced a feature-tagging approach to maintain trace links between source code and other software artifacts ubiquitously during software evolution [1, 40, 45].

4.4 Traceability Links Recovery Methods

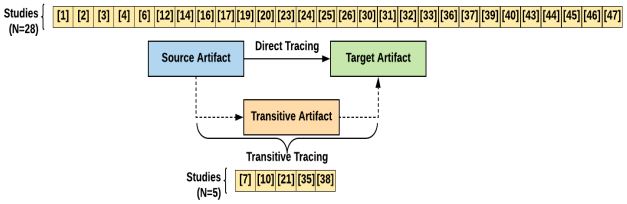

Figure 7: Traceability link recovery methods

Figure 7 illustrates the two types of traceability link–recovery methods used in the primary studies. The majority used direct tracing methods to create explicit trace links between source and target artifacts. We identified five studies [7, 10, 21, 35, 38] that used the transitive tracing approach, where traceability links two artifacts by joining them to a third.

In [10], Borg presented the transitive tracing approach to identify a set of potentially impacted artifacts for a new issue report. In this approach, the author used existing issue reports as the transitive artifacts to recover traceability links between a newly issued report and its related source code. The assumption is that the previous issue reports had more textual similarity relations with the current state of source-code artifacts than a new issue report. Similarly, Nishikawa et al.[35] proposed CLM, which establishes trace links between two artifacts by mapping transitively sourced artifacts to target artifacts via third artifacts, to overcome the textual similarity gap between source and target artifacts. The CLM approach extracts search terms from third artifacts to recover trace links between source and target artifacts.

Likewise, Berta et al.[7] used existing issue reports as transitive artifacts to recover traceability links between source code and use cases. They first extracted the lists of corresponding issue reports related to source code using source-code version history. Next, they calculated the textual similarity between issue reports and use cases to identify the impacted source code. Gharibi et al.[21] presented a similar approach that established traceability links between new feature requests and existing ones to identify the impact in source code. Similarly, Falessi et al.[17] established trace links between new and existing requirements to recover lists of impacted source-code classes. In [38], Rath et al. used existing requirements and similar bug reports as transitive artifacts to localise related source-code areas to fix new bug reports.

5 DISCUSSION

The discussion elaborates on the findings by grouping identified traceability link–recovery approaches according to our RQs. This study examined the literature concerning the use of traceability link–recovery approaches for CIA, noting the underlying challenges and limitations of the studies.

5.1 Findings of RQs

We found that four approaches (i.e., IR-based, heuristic-based, ML and DL) could be used to recover traceability links between software artifacts of different types. Most of the primary studies focused on enhancing IR-based approaches to identify outdated trace links and generate impact analysis reports. The VSM, LSI, JSM and LDA are the most popular IR models reported in the studies. No studies reported which IR models outperformed others. Approaches like ML and heuristics can predict the validity of candidate trace links generated with IR engines. Recently, some studies tried substituting IR-based approaches with DL to recover trace links. So far, one study [14] reported that word-embedding outperformed LSI in establishing trace links between test cases and source code. In terms of trace artifacts, we identified six artifacts (i.e., requirements, design, source code, test cases, bug reports and features). Among them, requirements, source code, and bug reports are the most frequently linked artifacts in the studies. Few studies focused on linking design and test case artifacts, presumably due to the scarcity of datasets. Recently, three studies [1, 40, 45] introduced feature artifacts as transitive artifacts to recover trace links between requirements and source code. In terms of trace direction, 31 of the 33 studies evaluated their approaches unidirectionally, as determining their approaches' bidirectional feasibility was difficult. In terms of tool support, all four tools were built on an IR-based approach;

only one supported a DL approach. In terms of evaluation, 30 of the 33 studies applied experimental research and evaluated their approaches with either open-source or university projects. Only three studies applied case studies and evaluated with industrial data. Due to data confidentiality, the context of the datasets was not provided in the details.

Our review highlighted that most studies focused on identifying program impact sets due to source code, making it the most frequent change area in software development. The studies used requirements, bug reports, test cases and features as source artifacts to identify program impact areas. Interestingly, design artifacts were left out of this impact set study. The second-most frequent studied area is the requirements impact set. The studies evaluated their approaches with either forward tracing (e.g., regulatory requirements to requirements) or backward tracing (e.g., source code to requirements). Test impact set studies followed the third positions and evaluated with source code, requirements and design artifacts. Only two studies[16, 35] focused on assessing design impact areas, both using requirements as source artifacts. Furthermore, only two studies[14, 35] used test cases as source artifacts and identified the corresponding source-code artifacts. Based on our findings, most studies followed the traditional software development approach and recovered links between requirements and source-code artifacts. Few studies focused on establishing links between test cases and source code, which is essential to support CIA in agile software projects [5, 13, 41].

This review also highlighted that trace links between artifacts of different types can be recovered either directly or transitively. Direct tracing is applicable to explicit tracing scenarios where the source and target artifacts share high textual similarity. Hence, direct tracing can recover trace links between existing artifacts of the system, whereas transitive tracing establishes trace links between new artifacts (e.g., a new feature request or bug report) and existing artifacts (e.g., source code). However, the challenging part of these approaches is finding the right transitive artifact for various CIA tasks. To the best of our knowledge, no traceability tool supports a transitive artifact approach, as it is challenging to extend research in this area.

5.2 Limitations and Future Work

Based on our findings, the studies suffered from the following gaps and challenges. Improvement in these areas is necessary to leverage automatic traceability-link recovery to support impact analysis effectively. To further advance automatic traceability research, we recommend these key improvements:

- Focus on tool enhancements to support fully automatic traceability-recovery approaches (e.g., machine learning based approach and deep learning based approach)
- Emphasise recovering links between trace artifacts commonly used in modern software development (e.g., user stories, accepted test cases and source code)
- Focus on building traceability systems beyond text-based recovery (e.g., recovering traceability links between design images and requirements)
- Investigating advanced static program analysis, such as value-flow analysis [42] and pointer aliasing analysis [29] to support more precise CIA.
- Evaluate industrial datasets and survey practitioners to gain valuable feedback for further improvements

6 THREATS TO VALIDITY

The following deviations from the study guidelines[28] may threaten this study's validity.

The first threat to validity is the selection of the studies and the relevance of review in the field. To mitigate this threat, we constructed our search strings by referring to previous review studies. We included all possible keywords to cover abbreviations, synonyms, morphological root forms (e.g., source code, architecture, trace*). We ran our search in five databases to cover a broader scope of concerns. To handle the threats of the relevance of study selection, the first author applied the selection criteria. The second and third authors validated 10% of the selected studies. The second threat is the reliability with which the authors carried out the data extraction, interpretation and findings justification. To eliminate this threat, the first author extracted data from all the selected publications. The second and third authors individually repeated this process for 20% of the selected studies. The authors then discussed the differences and reached the same conclusions.

The third threat is the scope of the review. Our review focused on automating traceability studies to assist the CIA process, as the scope is tight. We do not claim that our review applies to other areas of impact analysis (e.g., dependency impact analysis [8]). Thus, this external validity threat is minor. Furthermore, as the review protocol development presents in details in Section 3, other researchers can verify the validity of the findings with the search strategy, selection criteria and applied data extraction. Lastly, internal validity is concerned with the treatment and outcomes. As we presented our findings through a mixture of empirical studies and descriptive statistics, this threat is also minimal.

7 CONCLUSIONS

The paper presents the SLR on automatic traceability links recovery approaches to capture the current state of the literature related to CIA coverage. The review was conducted by following the guidelines provided by Evidence Based Software Engineering (EBSE) [28]. In total, we identified 33 studies and reviewed them in depth according to our research questions. The included studies were published between 2012 and 2019. Our review indicated very few traceability studies focused on design and test impact set analysis due to the scarcity of datasets. Based on the findings, we stress the need for industrial case studies. Finally, we presented suggestions to advance the traceability tool to support the latest traceability approaches, such as machine learning and deep learning.

8 ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their valuable comments. This research is supported by an Australian Research Grant DP200101328.

REFERENCES

[1] H. Abukwaik, A. Burger, B. K. Andam, and T. Berger. 2018. Semi-automated feature traceability with embedded annotations. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 529–533.

[2] N. Ali, H. Cai, A. Hamou-Lhadj, and J. Hassine. 2019. Exploiting parts-of-speech for effective automated requirements traceability. *Information and Software Technology* 106 (2019), 126–141.

[3] N. Ali, Y. Guéhéneuc, and G. Antoniol. 2013. Trustrace: Mining software repositories to improve the accuracy of requirement traceability links. *IEEE Transactions on Software Engineering* 39, 5 (2013), 725–741.

[4] G. Bavota, L. Colangelo, A. De Lucia, S. Fusco, R. Oliveto, and A. Panichella. 2012. TraceME: Traceability management in eclipse. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 642–645. https://doi.org/10.1109/ICSM.2012.6405343

[5] K. Beck. 2000. *Extreme programming explained: embrace change*. Addison-Wesley Professional.

[6] E. Ben Charrada, A. Koziolek, and M. Glinz. 2012. Identifying outdated requirements based on source code changes. In *2012 20th IEEE International Requirements Engineering Conference (RE)*. 61–70. https://doi.org/10.1109/RE.2012.6345840

[7] P. Berta, M. Bystrickỳ, M. Krempaskỳ, and V. Vranić. 2017. Employing issues and commits for in-code sentence based use case identification and remodularization. In *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*. ACM, 1.

[8] S. A. Bohner. 1996. Impact analysis in the software change process: a year 2000 perspective. In *1996 Proceedings of International Conference on Software Maintenance*. 42–51. https://doi.org/10.1109/ICSM.1996.564987

[9] S. A. Bohner and R. S. Arnold. 1991. Software change impact analysis for design evolution. In *Proceedings of the 8th International Conference on Software Maintenance and Reengineering*. 292–301.

[10] M. Borg. 2014. Embrace your issues: Compassing the software engineering landscape using bug reports. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 891–894.

[11] M. Borg, P. Runeson, and A. Ardö. 2014. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering* 19, 6 (2014), 1565–1616.

[12] O. Chaparro, J. M. Florez, and A. Marcus. 2017. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 376–387.

[13] J. Cleland-Huang. 2012. Traceability in agile projects. In *Software and Systems Traceability*. Springer, 265–275.

[14] V. Csuvik, A. Kicsi, and L. Vidács. 2019. Source code–level word embeddings in aiding semantic test-to-code traceability. In *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*. 29–36. https://doi.org/10.1109/SST.2019.00016

[15] A. De Lucia, F. Fasano, and R. Oliveto. 2008. Traceability management for impact analysis. In *2008 Frontiers of Software Maintenance*. 21–30. https://doi.org/10.1109/FOSM.2008.4659245

[16] D. Falessi, M. D. Penta, G. Canfora, and G. Cantone. 2017. Estimating the number of remaining links in traceability recovery. *Empirical Software Engineering* 22, 3 (2017), 996–1027.

[17] D. Falessi, J. Roll, J. L. Guo, and J. Cleland-Huang. 2018. Leveraging Historical Associations between Requirements and Source Code to Identify Impacted Classes. *IEEE Transactions on Software Engineering* (2018).

[18] F. Furtado and A. Zisman. 2016. Trace++: A traceability approach to support transitioning to agile software engineering. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*. 66–75. https://doi.org/10.1109/RE.2016.47

[19] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk. 2012. Integrated impact analysis for managing software changes. In *2012 34th International Conference on Software Engineering (ICSE)*. 430–440. https://doi.org/10.1109/ICSE.2012.6227172

[20] A. Ghannem, M.S. Hamdi, M. Kessentini, and H.H. Ammar. 2017. Search-based requirements traceability recovery: A multi-objective approach. In *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1183–1190.

[21] R. Gharibi, A. H. Rasekh, M. H. Sadreddini, and S. M. Fakhrahmad. 2018. Leveraging textual properties of bug reports to localize relevant source files. *Information Processing & Management* 54, 6 (2018), 1058–1076.

[22] O. C. Z. Gotel and C. W. Finkelstein. 1994. An analysis of the requirements traceability problem. In *Proceedings of IEEE International Conference on Requirements Engineering*. IEEE, 94–101.

[23] J. Guo, J. Cheng, and J. Cleland-Huang. 2017. Semantically enhanced software traceability using deep learning techniques. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 3–14.

[24] J. Guo, M. Gibiec, and J. Cleland-Huang. 2017. Tackling the term-mismatch problem in automated trace retrieval. *Empirical Software Engineering* 22, 3 (2017), 1103–1142.

[25] H. Hu X. Ma J. Lü P. Mäder H. Kuang, H. Gao and A. Egyed. 2019. Using Frugal User Feedback with Closeness Analysis on Code to Improve IR-Based Traceability Recovery *(ICPC '19)*. IEEE Press, 369–379. https://doi.org/10.1109/ICPC.2019.00055

[26] M. A. Javed and U. Zdun. 2014. A systematic literature review of traceability approaches between software architecture and source code. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 16.

[27] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. H. Hayes, A. Dekhtyar, D. Manukian, S. Hossein, and D. Hearn. 2012. TraceLab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *2012 34th International Conference on Software Engineering (ICSE)*. 1375–1378. https://doi.org/10.1109/ICSE.2012.6227244

[28] B. Kitchenham and S. Charters. 2007. Guidelines for performing systematic literature reviews in software engineering.

[29] Yuxiang Lei and Yulei Sui. 2019. Fast and precise handling of positive weight cycles for field-sensitive pointer analysis. In *International Static Analysis Symposium*. Springer, 27–47.

[30] A. Mahmoud and G. Williams. 2016. Detecting, classifying, and tracing non-functional software requirements. *Requirements Engineering* 21, 3 (2016), 357–381.

[31] T. Merten, D. Krämer, B. Mager, P. Schell, S. Bürsner, and B. Paech. 2016. Do information retrieval algorithms for automated traceability perform effectively on issue tracking system data?. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 45–62.

[32] C. Mills, J. Escobar-Avila, and S. Haiduc. 2018. Automatic traceability maintenance via machine learning classification. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 369–380. https://doi.org/10.1109/ICSME.2018.00045

[33] C. Mills and S. Haiduc. 2017. The impact of retrieval direction on IR-based traceability link recovery. In *2017 IEEE/ACM 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER)*. IEEE, 51–54.

[34] N. Mustafa and Y. Labiche. 2017. The Need for Traceability in Heterogeneous Systems: A systematic literature review. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 305–310.

[35] K. Nishikawa, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe. 2015. Recovering transitive traceability links among software artifacts. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 576–580.

[36] A. Panichella, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. 2013. How to Effectively Use Topic Models for Software Engineering Tasks? An Approach Based on Genetic Algorithms *(ICSE '13)*. IEEE Press, 522–531.

[37] M. Rahimi and J. Cleland-Huang. 2018. Evolving software trace links between requirements and source code. *Empirical Software Engineering* 23, 4 (2018), 2198–2231.

[38] M. Rath, D. Lo, and P. Mäder. 2018. Analyzing requirements and traceability information to improve bug localization. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 442–453.

[39] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder. 2018. Traceability in the wild: automatically augmenting incomplete trace links. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 834–845.

[40] M. Seiler, P. Hübner, and B. Paech. 2019. Comparing traceability through information retrieval, commits, interaction logs, and tags. In *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*. 21–28. https://doi.org/10.1109/SST.2019.00015

[41] T. Stålhane, G. K. Hanssen, T. Myklebust, and B. Haugset. 2014. Agile change impact analysis of safety critical software. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 444–454.

[42] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. 265–266.

[43] R. Tsuchiya, H. Washizaki, Y. Fukazawa, K. Oshima, and R. Mibe. 2015. Interactive recovery of requirements traceability links using user feedback and configuration management logs. In *International Conference on Advanced Information Systems Engineering*. Springer, 247–262.

[44] M. Unterkalmsteiner, T. Gorschek, R. Feldt, and N. Lavesson. 2016. Large-scale information retrieval in software engineering-an experience report from industrial application. *Empirical Software Engineering* 21, 6 (2016), 2324–2365.

[45] T. Vale and E. S. D. Almeida. 2019. Experimenting with information retrieval methods in the recovery of feature-code SPL traces. *Empirical Software Engineering* 24, 3 (2019), 1328–1368.

[46] W. Wang, A. Gupta, N. Niu, L. D. Xu, J.-R. C. Cheng, and Z. Niu. 2016. Automatically tracing dependability requirements via term-based relevance feedback. *IEEE Transactions on Industrial Informatics* 14, 1 (2016), 342–349.

[47] W. Wang, N. Niu, H. Liu, and Z. Niu. 2018. Enhancing automated requirements traceability by resolving polysemy. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 40–51.