A Systematic Literature Review of Traceability Approaches between Software Architecture and Source Code

Muhammad Atif Javed and Uwe Zdun
Software Architecture Research Group
University of Vienna, Austria
muhammad.atif.javed|uwe.zdun@univie.ac.at

ABSTRACT

The links between the software architecture and the source code of a software system should be based on solid traceability mechanisms in order to effectively perform quality control and maintenance of the software system. There are several primary studies on traceability between software architecture and source code but so far no systematic literature review (SLR) has been undertaken. This study presents an SLR which has been carried out to discover the existing traceability approaches and tools between software architecture and source code, as well as the empirical evidence for these approaches, their benefits and liabilities, their relations to software architecture understanding, and issues, barriers, and challenges of the approaches. In our SLR the ACM Guide to Computing Literature has been electronically searched to accumulate the biggest share of relevant scientific bibliographic citations from the major publishers in computing. The search strategy identified 742 citations, out of which 11 have been included in our study, dated from 1999 to July, 2013, after applying our inclusion and exclusion criteria. Our SLR resulted in the identification of the current state-of-the-art of traceability approaches and tools between software architecture and source code, as well as gaps and pointers for further research. Moreover, the classification scheme developed in this paper can serve as a guide for researchers and practitioners to find a specific approach or set of approaches that is of interest to them.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: Surveys of historical development of one particular area; D.2.7 [**Software Engineering**]: Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Management, Documentation

Keywords

Traceability, Software Architecture, Source Code, Systematic Literature Review

1. INTRODUCTION

Traceability between the architecture and the source code of a software system helps developers in easier understanding architecture designs and provide support for software quality control and maintenance. An important problem in this context is that software systems evolve over time, including the architecture and the source code. This might cause serious problems, as during evolution the consistency of the traceability links between the architecture and the source code is continuously broken. The separate evolution of these artefacts might cause mistrust in existing traceability information, and, as a result, the traceability information becomes obsolete. In addition, it is hard to find the correct links between architecture and implementation artefacts because usually software architecture is not explicitly represented in the source code, e.g. as packages and classes or similar code-level abstractions. Traceability approaches aim to provide means to cope with these challenges, in particular, to support the rapid identification of architecture artefacts in the source code and to assure consistency between them.

There are several primary studies on traceability approaches between software architecture and the source code, but so far no Systematic Literature Review (SLR) has been undertaken. This study presents the first systematic literature review on traceability approaches and tools between software architecture and source code. The goal of this study is to systematically access and examine the existing studies on traceability between software architecture and the source code by following the guidelines of Kitchenham et al. [10, 11]. To do so, the Association for Computing Machinery (ACM) Guide to Computing Literature has been electronically searched to accumulate the biggest share of relevant scientific bibliographic citations from the major publishers in computing. Using our search strategy we have identified 742 citations, out of which 11 are selected for our SLR on traceability between software architecture and the source code.

The result of our SLR is the identification of the present state of traceability approaches and tools between software architecture and the source code. We have studied the existing empirical evidence for these approaches, their benefits and liabilities, their relations to software architecture understanding, and issues, barriers, and challenges of the

approaches. In addition, a classification scheme has developed. It distinguishes various aspects of traceability between software architecture and the source code, including the nature of the approaches, automation of traceability approaches, types of traceability relations, granularity of traceability approaches, traceability direction, and representation of traceability information. This categorization of the various aspects within the field of traceability between software architecture and the source code can serve as a guide for researchers and practitioners to identify a specific set of approaches that is of interest to them.

This paper is organized into five sections: The following Section 2 discusses the systematic literature review process we followed. In Section 3 we present the results from the analysis of our systematic literature review. Section 4 discusses the results of our SLR in relation to the research questions addressed in the SLR. Finally, in Section 5 we conclude and discuss future work.

2. SYSTEMATIC LITERATURE REVIEW PROCESS

An SLR, also referred to as systematic review or literature review, is a well-defined and rigorous method for identifying, evaluating, and interpreting all available research relevant to a particular research question, topic area, or phenomenon of interest [10]. SLRs aim to present a fair, unbiased, and credible evaluation of a particular research topic by utilizing auditable, rigorous, and trustworthy methods [10].

This research has been conducted as an SLR by following the guidelines of Kitchenham et al. [10, 11], who propose three main phases:

1. Planning the review, which aims at identifying the main rationale for undertaking the review and developing a review protocol (see Section 2.1),

2. executing the review, which aims at conducting the review by executing the planned review protocol from the previous phase (see Section 2.2), and

3. reporting the review, which aims at presenting the results of the review and its dissemination to the interested parties (see Sections 3 and 4).

The objective of this review is to identify the state-of-the-art on traceability approaches between software architecture and the source code during the years 1999 – July, 2013. To select this particular period, we have performed a thorough overview research before the study, where we found that this particular period covers the most significant approaches in the area of software architecture traceability.

2.1 Planning the review

The first phase of an SLR is associated with pre-review activities for conducting the systematic review. Steps involved in planning the review are identification of the need for a review (see Section 2.1.1), specifying the search questions (see Section 2.1.2), the search strategy (see Section 2.1.3), and inclusion and exclusion criteria (see Section 2.1.4).

2.1.1 Identification of the need for review

There are several primary studies on traceability between software architecture and the source code (such as [1, 2, 4, 8, 9, 17, 19, 21, 23, 24]), but so far no systematic literature review has been undertaken. Therefore, this study is conducted as the first systematic literature review to summarise all existing practices and information with regard to the relationship of traceability between software architecture and the source code in a thorough and unbiased manner. Moreover, this review is a first step towards drawing more general conclusions, more specifically, on the role of traceability in understanding of software architectures. Accordingly, this review provides a resource for practitioners (to select the appropriate traceability method) and for researchers (to determine the gaps and pointers for further research) in this area.

2.1.2 Specifying the research question

The research question specification step is considered as one of the most important parts of an SLR, because the research questions drive the entire literature review methodology. The selected research questions for achieving the outlined objectives of our study are listed below:

1. What is the state-of-the-art in traceability approaches and tools between software architecture and the source code?

2. What information is available for traceability from architectural artifacts to more low-level artifacts (like the source code) and vice versa?

3. What empirical evidence has been reported in the field of traceability between software architecture and the source code?

4. In how far are the reported traceability relationships useful in software architecture understanding?

5. What are the reported benefits and liabilities of traceability approaches between software architecture and the source code?

6. What are the reported issues, barriers, and challenges of traceability between software architecture and the source code?

2.1.3 Search Strategy

The search strategy step of an SLR aims to provide a foundation for the comprehensive and unbiased collection of research works from the literature. This requires a well-planned search strategy so that every relevant bibliographic citation has a very good chance to appear in the search results.

The ACM digital library[1] aims to provide a comprehensive coverage of bibliographic citations, which are published by ACM and a wide range of publishers in the field of computing. The methodology, which we exploited for reaching comprehensive coverage, is to perform an advanced search in the ACM Guide to Computing Literature for accumulating bibliographic citations from the major publishers in computing. At the time of our study, the ACM Guide to Computing Literature contained approximately two million bibliographic records. The rationale behind using this particular methodology is to find the biggest share of scientific citations that are relevant to answer the specific research questions enlisted above.

---

[1]http://dl.acm.org/

The search string comprises keywords derived from the research topic and Boolean logic for performing efficient searching. Consequently, a citation is available in the list of search results upon matching of keywords and Boolean logic. The following search string is created to search for relevant publications:

```
(((((Abstract:"architecture") or (Abstract:"architectures") or
  (Abstract:"architectural") or (Abstract:"model") or
  (Abstract:"models") or (Abstract:"design") or
  (Abstract:"designs") or (Abstract:"component") or
  (Abstract:"components"))
 and ((Abstract:"code") or (Abstract: "implementation")))
 and ((Abstract:"trace links") or (Abstract:"tracing") or
    (Abstract:"traceability")))
```

However, this process might inevitably miss useful citations. In response to this problem, snowballing [5] (i.e., following the references of articles found during the search) is performed in an iterative way to identify particular citations. That is, bibliographies of every selected citation are checked for useful articles that are missed in the initial search. Snowballing is performed until a convergence is reached and no more new articles are found.

2.1.4 Inclusion and Exclusion Criteria

Inclusion and exclusion criteria are used to determine the suitability of publications (articles) and making decisions about whether to accept or reject a particular publication into the SLR. The inclusion and exclusion criteria for this SLR are listed below:

Inclusion Criteria

- Studies on traceability approaches or tools that are related to the software architecture and the source code.

- Studies that discuss the effects of traceability in architecture or design models and the source code.

- Studies that provide some sort of solution, roadmap, or framework related to traceability between software architecture or design models and the source code.

- Studies that report success, issues, and/or failures or any type of experience concerning traceability between architecture or design models and the source code.

- Studies representing empirical evidence on traceability between software architecture or design models and the source code by describing experiments, surveys, experience reports, case studies, or observation articles.

- Studies published in peer-reviewed journals, conferences, and workshops.

Exclusion Criteria

- Studies reporting information on traceability (only) in hardware architectures, requirements, low-level software design, or other fields not directly related to traceability between software architecture and the source code.

- Studies related to software architectures and the source code, but not or only very marginally concerning traceability.

- Every study that is not published in a peer-reviewed journal, conference, or workshop is excluded. For instance, books, master thesis, and doctoral dissertations are excluded to establish quality through peer review.

- Studies published outside of the time-frame 1999 – July, 2013 (the initial search of this SLR has been performed in August, 2013). To select this particular period, we have performed a thorough overview research before the study, where we found that this particular period covers the most significant research in the area of software architecture traceability.

- Studies having more than one published descriptions are included only once (using the most detailed and up-to-date version of the study).

2.2 Conducting the Review

The second phase of the SLR method aims at conducting the SLR. Steps involved in conducting the SLR are search for studies and studies selection (see Section 2.2.1), data extraction (see Section 2.2.2), and study quality assessment (see Section 2.2.3).

2.2.1 Search for Studies and Studies Selection

To create the initial pool of articles, automatic search for citations was performed without any additional constraints. The search was conducted in August, 2013 using an advanced search in the ACM Guide to Computing Literature with the search string described above. Overall 726 citations have been identified in this first step using the search strategy.

After obtaining this initial body of literature, we reviewed each of the citations. In this process, the first author selected the set of citations which could be relevant by reading the abstracts of the citations as well as having a cursory looking into the citations in case it was hard to predict from the abstract if the citation shall be included or not. This selection is based on the inclusion and exclusion criteria explained above. The second author independently checked all citations, and in cases of doubt we discussed the citation in question in detail. Finally, both authors read all selected publications in detail. 11 citations conformed to the inclusion criteria and have therefore been included in the study.

In the following snowballing process, for each selected citation, a secondary search was performed based on the references of that particular citation. For the citations selected, we applied the same studies selection process again. Snowballing continued until no more new citations were found, leading to 16 additional citations included in our SLR. Some of the articles found during snowballing did not exist in the ACM digital library. Others did exist in it, but were not found in the first search because they have very short abstracts or simply their abstracts do not include at least one word from each of the three parts of the search string. None of those 16 additional citations conformed to the inclusion criteria. Hence, overall we selected 11 studies out of a total of 742 citations.

2.2.2 Data Extraction

In this SLR the following form is used to extract the data for addressing the research questions from all included studies.

1. Reference of bibliographic citation (including title, author(s) name, conference proceedings or journal, date of publication)

2. Specific information concerning software architecture traceability:

   (a) Challenges or problems

   (b) Approach (e.g., available in literature or proposed by authors)

   (c) Tool (e.g., open source, commercial, or proposed by authors)

   (d) Automation of traceability approach

   (e) Types of traceability relations

   (f) Granularity of traceability approach

   (g) Traceability direction

   (h) Representation of traceability information

   (i) Comparison (benefits and drawbacks of the particular approach)

   (j) Arguments for supporting architectural understandability

   (k) Form of empirical evidence

3. Validity threats (internal or external)

2.2.3 Quality Assessment

The quality assessment criteria are used to examine the rigour and credibility of the used research methods as well as the relevance of the citations. The following quality criteria have been applied:

1. Are the collected citations relevant? In this SLR, the quality assessment is specifically focused on accumulating only those citations that report adequate information to answer the stated research questions. The quality assessment has been performed according to our inclusion/exclusion criteria by the two researchers who have independently reviewed each citation in a number of iterations.

2. Does the literature research cover all relevant citations? In this SLR, the two researchers have independently reviewed each citation in a number of iterations to ensure that none of the relevant citations are missed. The steps toward achieving this objective are to search from the entire population of scientific citations and screening by title, abstract, keywords, and conclusion after accessing the full text.

3. Do the citations contain adequate information? In this SLR, it is examined whether particular citations contain sufficient information to answer the research questions and whether the particular work is supported by some (preferably empirical) evidence.

3. RESULTS FROM THE SYSTEMATIC LITERATURE REVIEW

This section presents the results of our SLR on traceability approaches between software architecture and the source code. It aims at collecting and summarizing the results of the included primary studies and provides a comprehensive and orthogonal classification of the research results in these studies. Six main categories have been identified and used to distinguish the different research approaches: Traceability approaches between software architecture and the source code (see Section 3.1), automation of traceability approaches (see Section 3.2), types of traceability relations (see Section 3.3), granularity of traceability approaches (see Section 3.4), traceability direction (see Section 3.5), and representation of traceability information (see Section 3.6).

3.1 Traceability Approaches between Software Architecture and the Source Base

3.1.1 Event-Based Traceability

Buchgeher and Weinreich [4] introduce the LISA approach to semi-automatically capture traceability relations from an architectural component model to the code base, design decisions, and architecture-significant requirements. The LISA toolkit is based on a semi-formal architecture description model, which is continuously synchronized with the system implementation. LISA supports automatically establishing traceability relations through observing the developer as she/he is working on the architecture design and implementation. Traceability is captured in three steps: In the first step, the developer specifies the context of the work by selecting the active design decisions. In the next step, the developer performs architecture design and implementation tasks. During these tasks, modification events are created and logged. A modification event contains short descriptions of the performed modification, potential targets of traces, and manipulated architecture or implementation elements (which are affected by the architecture design or implementation activity). The modification request is sent to an event-logger which is in charge of capturing the trace targets and managing the active decisions. In the last step, a review of the traceability target is performed. The approach is validated using an action research approach.

Traceability information requires maintenance as the related development artefacts evolve. **Mäder et al.** [13] introduce a semi-automatic strategy to determine changes in Unified Modelling Language (UML) models to update pre-existing traceability links. The approach records all changes to model elements and uses this information to find a match within a set of predefined patterns of recurring development activities. A match will trigger the required traceability update actions. While the approach is not only related to architectural models, it can be used for maintaining traceability between architecture and design models, as it has been defined for various structural UML diagrams (class, object, composite structure, package, and component diagrams). The usefulness of the approach is shown through an experiment that compares the effort and quality associated with and without supported tool, called traceMaintainer.

Hammad et al. [9] introduce an approach that supports traceability from the code base to the design to maintain consistency with the design during code evolution. The approach examines the code base changes based on a lightweight analysis and syntactic differentiation to evaluate whether a particular change alters the design or not. The changes in the code base that lead to design changes include adding or removing of classes, methods, and their relationships. In contrast, modifications in data structures, loops, or conditional statements typically do not lead to

changes in the design. The outcome contains details of changes and their particular impacts on the design, which are reported to the user upon identification of a design change. The usefulness of the approach is shown through an experiment that compares the accuracy and time spent with and without supported-tool, called SrcTracer.

3.1.2 Rule-Based Traceability

Murta et al. [18, 19] propose ArchTrace, an architecture-implementation traceability approach that focusses on keeping an evolving conceptual architecture synchronized with its evolving code base. ArchTrace uses policies, specified by the architects and developers, to describe how to continuously update traceability links between the architecture and its implementation. That is, in response to each and every change in an architecture or source code artifact, one or more policies might be triggered, which manage the evolution of traceability links. A policy can be a rule or constraint that is used to determine or limit the set of actions to be applied for updating traceability links. The policies make use of encoded knowledge (concerning architecture and source code artifacts) for making educated guesses. ArchTrace also supports the visualization of traceability links. The approach is evaluated through a retrospective study.

3.1.3 Hypertext-Based Traceability

Alves-Foss et al. [1] introduce an XML-based traceability approach to support traceability between a UML-based design specification and the corresponding code base. In this approach, XML Metadata Interchange (XMI)[2] and JavaML [14] are used as markup specifications of the design and the code base respectively. The approach uses Xlink-base[3], a linking language, to specify traceability links that describe which design classes are implemented by which code base classes and vice versa. The Extensible Stylesheet Language (XSL)[4] is used to render the XMI design specification, the JavaML code markup, and the traceability links in HTML. The approach is evaluated through a case study.

Nguyen [20, 21] introduces MolhadoArch, a hypertext versioning and software configuration management approach to consistently maintain architecture configurations and versions at various levels of abstraction, including architecture elements, the code base, and traceability relations between them. In MolhadoArch, the architecture is modelled as an attributed graph, which incorporates five entities, namely, component, atomic component, composite component, connector, and interface, to precisely capture architecture elements and their interconnections. Every atomic or composite component is represented as a node, which is further associated with referential attributes (or components) that hold a reference to the corresponding component. MolhadoArch represents the code base in form of a hierarchical structure, the code base tree, where each tree node represents a structural unit in the code. Each traceability relationship is represented by means of an edge between the nodes in the graph and the corresponding node in tree data structure. MolhadoArch requires human effort or trace generation tools to establish the traceability relations. It supports evolution by capturing the state of the project at various discrete points

in time that can be retrieved and used in later sessions. The usefulness of the approach is evaluated through an experiment that compares the performance and efficiency of MolhadoArch.

3.1.4 Traceability Based on Information Retrieval

The design or architecture of a software systems can be reconstructed from the code base using reverse engineering mechanisms. In general, the continuous evolution of a design is preferred over design or architecture reconstruction, because it contains richer information and higher quality, compared to reconstructed architectures or designs [2, 9]. **Antoniol et al.** [2] introduce an evolutionary design-implementation traceability approach and a tool to check the compliance of design and code base. The approach exploits reverse engineering mechanisms to extract the design from the code base. During the reverse engineering process, commonalities and differences between the design and the code base are identified. Commonalities are identified by exploiting edit distance of attribute names and the maximum matching algorithm [6]. The maximum matching algorithm is used to determine traceability links using the average similarity and differences (unmatched attributes) based on the extracted edges, selected edges, and missed edges respectively. In addition, the extracted design is compared with the actual design to better cope with inconsistencies. The resulting design is graphically visualized with green, yellow, and red colours to support better understandability of perfect match, poor match, and unmatched classes, respectively. The usefulness of the approach is experimented on industrial design and code.

3.1.5 Design Patterns Based Traceability

Ubayashi and Kamei [24] use the concepts of Archpoints (architecture points) and Archmappings (architecture mappings) to support traceability between the architecture and the code base. Archpoints are selected points in the architecture that describe the essence of an architectural design with respect to structural and behavioural aspects. Traceability is established by mapping them to program points, which is implemented using data flow analysis based on the Observer design pattern [7]. For instance, an archpoint such as 'message send' in the design is mapped to a program point such as 'method call' in the code. Archmappings are used to validate traceability between an architectural design and the code base. Constraints (such as, 'execution_order') are converted into logical formulas to verify the traceability (by determining the satisfiability of logical formulas). For this, an SMT (Satisfiability Modulo Theories) solver, a tool for deciding the satisfiability of logical formulas, is used.

3.1.6 Model-driven Traceability

Tran et al. [23] present a view-based model-driven traceability framework (VBTrace): The goal of VBTrace is to support modelling and development of service-oriented systems at various levels of granularity and abstraction based on the particular needs, knowledge, and experience of stakeholders. The VBTrace tool uses code generation templates to automatically establish traceability links between all design views, as well as between high-level and low-level representations of the views and the code base. VBTrace supports both forward and reverse engineering of

traceability links: Firstly, it supports forward engineering through code generators that generate both code and traceability links from the view models. Secondly, it supports reverse engineering through view-based interpreters for reconstructing the traceability relations between architecture/design views and the code base. The approach is evaluated through an industrial case study.

Haitzer and Zdun [8] introduce a semi-automatic strategy to support the traceability between architectural models and the source code. The goal of the approach is to provide architectural abstraction specifications in a domain specific language (DSL) that only requires changes, if changes in the code base lead to architecture changes, but tolerate non-architectural changes. In particular, the approach comprises two main steps to set up the traceability links. In the first step, a class model is extracted from the source code – in order to decouple the approach from a specific source language. In the second step, an architectural abstraction specification is defined in a DSL to describe which parts of the source code contribute to a specific component. This is supported through multiple rule-based filters and group clauses. Based on the architectural abstraction specification, defined in the DSL code, the approach automatically generates the UML component model from source code using model transformations and supports the traceability between the mapped artifacts. The approach also supports the comparison of a generated component model with the actual or previous component model to better cope with inconsistencies. The approach is evaluated through a case study.

3.1.7 Traceability Based on the Machine Learning Techniques

An architectural tactic is a means of satisfying a response measure on a quality attribute (such as performance, reliability, usability, or maintainability) by manipulating some aspect of a quality attribute model through architectural design decisions [3]. For instance, an example of a performance tactic is 'reduce the computational overhead'. **Mirakhorli and Cleland-Huang [17]** propose an tactic-centric approach to support the automated reconstruction of traceability links between code classes and architectural tactics. It builds upon the fundamental concept of tactic traceability information models (TTIMs) that comprises of tactic-specific requirements, quality goals, rationales, roles, and proxies [16]. The approach utilizes information retrieval and machine learning methods to train a classifier (e.g., training with tactic descriptions and code snippets), to detect the tactic-related classes. The approach also uses the lightweight structural analysis to identify the role-related classes from the detected tactic-related classes, which clearly defines tactic's roles. The identified tactic-related and role-related classes are mapped to the proxies in TTIMs to generate the tactic-level traceability links. The approach is evaluated through a case study.

3.2 Automation of Traceability Approaches

Traceability approaches and tools offer varying levels of automation. In the literature we have found three main levels of the automation of traceability approaches, ranging from manual over semi-automatic to automatic establishment and maintenance of traceability relations, as shown in Table 1. In this section, those levels are used to distinguish the different research approaches.

	Nature of Traceability	Automation	Tool-Support	Evidence
Alves-Foss et al. [1]	Hypertext-Based Traceability	Manual	ArgoUML	Case Study
Antoniol et al. [2]	Information Retrieval-Based Traceability	Semi-Automatic	Compliance Checker Tool	Experiment
Buchgeher and Weinreich [4]	Event-Based Traceability	Semi-Automatic	LISA Toolkit	Action Research
Haitzer and Zdun [8]	Model-driven Traceability	Semi-Automatic	DSL-Based Tool	Case Study
Hammad et al. [9]	Event-Based Traceability	Semi-Automatic	SrcTracer	Experiment
Mäder et al. [13]	Event-Based Traceability	Semi-Automatic	TraceMaintainer	Experiment
Mirakhorli and Cleland-Huang [17]	Traceability Based on the Machine Learning Techniques	Automatic	Tactics-Based Tool	Case Study
Murta et al. [18, 19]	Rule-Based Traceability	Automatic	ArchTrace	Retrospective Study
Nguyen [20, 21]	Hypertext-Based Traceability	Semi-Automatic	MolhadoArch	Experiment
Tran et al. [23]	Model-driven Traceability	Semi-Automatic	VBTrace	Case Study
Ubayashi and Kamei [24]	Design Patterns Based Traceability	Manual	SMT-Based Tool	Not Provided

Table 1: Nature, Automation, Tool-Support, and Evidence of the Approaches

3.2.1 Manual

The first group of approaches requires manual effort (by a human) for establishing and maintaining traceability relations. We identified two approaches [1, 24] that require human effort for establishing and maintaining traceability relations. Out of those approaches, one approach [24] only support automatic extraction and analysis of pre-specified traceability information, whereas the other one [1] uses various XML representations to describe the traceability information. The former is partially supported by a SMT-Based tool[5], while latter use ArgoUML, an open source UML modeling tool to support the hypertext-based traceability.

Although, manual traceability tools support developers in the management of traceability relations and minimize the time needed to find the desired links, the benefits of traceability are reduced due to the enormous effort for creation and maintenance of traceability relations, even in the presence of qualified developers and strong management [22]. That is, manual creation and maintenance of traceability relations is difficult, time consuming, complex, error-prone, and costly. However, manual traceability is useful in cases where only a limited number of specific traceability links must be specified that are difficult to automate and/or when human judgment is needed for each traceability link that should be established.

3.2.2 Semi-Automatic

The second group of approaches proposes automated methods and tools together with human activities for semi-automatic establishing and maintaining of traceability relations. These approaches require human analysts to monitor the results produced by automated methods and to interact with the supported tool to provide feedback and retrace. Seven of the approaches [2, 4, 8, 9, 13, 21, 23] in our SLR support semi-automatic traceability. Out of those approaches, one approach [9] inform the human analyst about the impacts of observed changes and require support for making certain decisions; the approach is

---

[5]http://argouml.tigris.org/

	Dependency	Generalization or Refinement	Evolution	Satisfiability	Overlap	Conflict	Rationalization	Contribution
Alves-Foss et al. [1]	The approach uses Xlink-base, a linking language, to specify the *design-to-sourcecode-links*							
Antoniol et al. [2]		✓	✓	✓				
Buchgeher and Weinreich [4]	✓	✓		✓	✓	✓	✓	✓
Haitzer and Zdun [8]	✓							
Hammad et al. [9]	✓	✓	✓					
Mäder et al. [13]	✓	✓	✓					
Mirakhorli and Cleland-Huang [17]	✓				✓	✓	✓	✓
Murta et al. [18, 19]	✓	✓	✓		✓	✓		
Nguyen [20, 21]	✓	✓	✓					
Tran et al. [23]	✓	✓		✓		✓	✓	
Ubayashi and Kamei [24]	✓			✓				

Table 2: Overview of traceability relationship types found in the literature

supported by the SrcTracer tool. Two of the tool-supported approaches [13, 21], named as TraceMaintainer and MolhadoArch, respectively, supports automated maintenance of pre-existing traceability links. One approach [23] supports semi-automatically eliciting and formalizing the trace dependencies, and two other approaches [4, 8] support semi-automatically capturing the traceability relations; the approaches are supported by the VBTrace, LISA Toolkit and DSL-Based Tool, respectively. Finally, one tool-supported approach [2] recovers the design from the code-base and compares it with the actual design, to help users in better coping with inconsistencies.

3.2.3 Automatic

The group of automatic traceability approaches generate traceability relations as a result of the software development process. They support automated establishing and maintaining of traceability relations. We identified two approaches [19, 17] that supports automated establishing of traceability; the approaches are supported by the ArchTrace and Tactis-Based Tool, respectively. The approaches, however, require initial traceability links from the developers.

Even though automated traceability is much faster than manual and semi-automatic traceability, it might either produce inaccurate traces or miss traces, and tends to result in a low precision and proliferation of traceability links that are difficult to manage and understand [15].
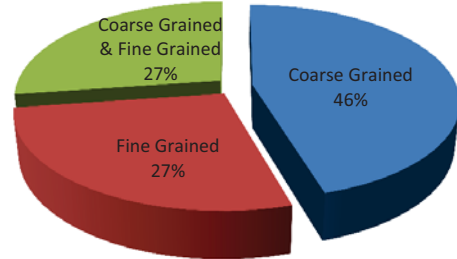
3.3 Types of Traceability Relations

Traceability approaches and tools exploit various kinds of relations to represent traceability information with respect to the different interests, goals, and perspectives of the stakeholders. To this end, it should be noted that understanding, interpreting, and analysing these relations relies on the stakeholder needs, knowledge, and experience [12]. Table 2 shows that existing traceability approaches use dependency, generalization and refinement, evolution, satisfiability, over-

lap, conflict, rationalization, and contribution relations to interrelate software architecture and the source code.

3.4 Granularity of Traceability Approaches

Traceability approaches and tools establish links between development artefacts at varying levels of granularity. In our SLR, two main levels of the granularity, coarse-grained and fine-grained, have been identified and used to distinguish the different research approaches. The distribution of the studies by levels of the granularity is shown in Figure 1.
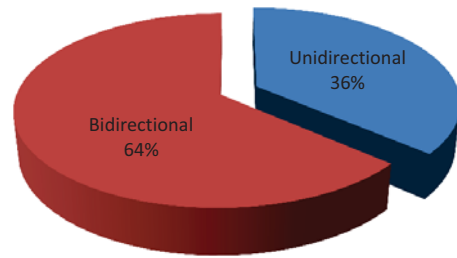

Figure 1: Distribution of the Studies by Levels of the Granularity

In the group with **coarse-grained** granularity, traceability relations are established between coarse-grained artefacts, for instance, components, classes, etc. In the literature, we identified 8 studies [1, 2, 8, 9, 13, 17, 19, 23] that support coarse-grained links.

In the group with **fine-grained** granularity, traceability relations are established between fine-grained elements, such as attributes, functions, parameters, to provide the maximum support of traceability. In the set of studies supporting fine-grained traceability, six studies [4, 9, 13, 17, 21, 24] support fine-grained traceability between architecture elements and implementation methods. It is interesting to note that three of the fine-grained approaches [9, 13, 17] also support coarse-grained links.

3.5 Traceability Direction

Traceability approaches and tools offer unidirectional or bidirectional specification, maintenance, or representation of traceability information. The distribution of the studies by traceability direction is shown in Figure 2.

Figure 2: Distribution of the Studies by Traceability Direction

The group of **unidirectional traceability** approaches supports to establish 'trace to' links from one artefact to another (also called forward traceability). In the literature, we identified four studies that support unidirectional traceability links between architecture and the code base: Three

studies [2, 8, 9] support traceability from the code base to the architecture (such as classes, interfaces, relationships, etc.), whereas the other one [17] supports traceability from the code base to architectural tactics.

The group of **bidirectional traceability** approaches supports forward as well as backward traceability. Backward traceability supports developers and maintenances to determine the origin of the traceability link. We identified 7 approaches [1, 4, 13, 19, 21, 23, 24] that support bidirectional traceability links between architecture and the code base. Five of those approaches [1, 4, 13, 19, 23] support traceability between architectural models and code elements. One approach [24] supports traceability between architecture points and program points, and one other approach [21] maintains traceability between an attributed graph and the code base tree.

3.6 Representation of Traceability Information

The representation criterion observes the main structures used for tracking and understanding the traceability information. In the literature, five main categories of the representation of traceability information, matrices, lists, hyperlinks, models, and graphs, have been identified, the distribution of the studies is shown in Figure 3:

Matrix: A traceability matrix is a two-dimensional grid which represents traceability links between two sets of artifacts, such as architecture, code elements, etc. The rows and columns of the grid are associated with the artifacts, and the grid-intersection points represent the existence of a link. One of the studies [17] in our SLR uses traceability matrices to represent the traceability information between architectural tactics/tactics-roles and code classes.

Lists: Traceability relations can also be represented as lists. Each entry in a list contains information about source, target artifacts and other attributes, which represent individual traceability link. We identified three studies that use lists. One of the study [9] produces the details of changes and their particular impacts on the design. One other study [24] suggests the use of Yices SMT-solver for deciding the satisfiability of logical formulas. The Yices SMT-solver generates counterexamples as output. Finally, one study [19] produces the information about created traceability links, the messages posted by policies in ArchTrace, which contain timestamp, source and target artifacts, and their versions.

Hyperlinks: Hyperlinks can be used to connect related concepts, keywords, or phrases in a natural way. The hyperlinks enable users while browing an artifact to easily 'jump' to another artifact. Two of the studies [1, 21] in our SLR provide hyperlinks support navigating from source code to design and vice-versa.

Graphs: Graphs support visualization of multidimensional traceability relationships between architecture and code artifacts by representing artifacts as nodes and relationships between them as edges. Thus, two nodes in the graph are connected if traceability link exists between the corresponding artifacts. One of the studies [13] in our SLR use arcs to represent the traceability relations between the nodes of the graph; the nodes represent related elements in same or different models. One other study [21] represents traceability relationship by means of an edge between the nodes in the graph, represents architecture elements and their interconnections, and the corresponding node in the

code base tree. Finally, one approach [2] uses graph visualizations to show the results of matching, highlighting similarities and differences between classes in the design and in the code.

Traceability Models: Traceability information can be accessed with traceability model. A model provides guidance about which artifacts to collect and which relations to establish in order to meet the traceability goals. Thus, the cornerstone of traceability framework is a traceability model. Three of the studies in our SLR use models to represent the traceability information. One of the study [23] illustrates the trace dependencies in the 'Traceability Model' and the model-based generated code. One other study [4] uses annotated traces with architecture models and code editors to represent the traceability information. Finally, one study [8] represents traceability as a model specified in a DSL to link to UML class and component models.
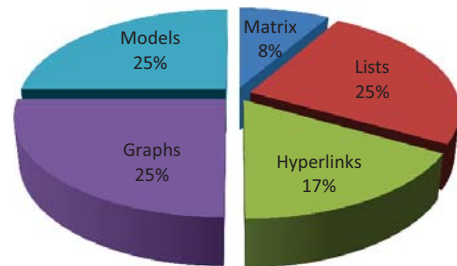

Figure 3: Distribution of the Studies by Representation of Traceability Information

4. DISCUSSION

Our SLR resulted in the selection of 11 unique traceability approaches between software architecture and the source code for the literature survey. The findings shows that the years 1999 – July, 2013 cover the most significant traceability approaches between software architecture and the source code, with a strong positive upward trend during the years 2011 – 2012, the particular period covers 54.5% of the selected studies. Hence, we can conclude that the research in the area of traceability between software architecture and the source code is increasing. To provide an overview of the current state-of-the-art on traceability approaches between software architecture and the source code and to guide the readers to a specific set of approaches that is of interest to them, the studies in the different categories (nature of the approaches, automation of traceability approaches, types of traceability relations, granularity of traceability approaches, traceability direction, and representation of traceability information) are discussed in Section 3 in detail.

With regard to tool support, we found that all of the investigated primary studies are supported by the traceability tools. Eight of the approaches are supported by complete toolkits, while two approaches are partially supported by the tools. Lastly, one approach uses ArgoUML, an open source UML modeling tool to support hypertext-based traceability. We have further found that the large majority of these tools supports semi-automatic traceability. Please note that tools supporting manual and automated traceability are typically not recommended for industrial settings due to the higher effort in case of manual approaches or lower accuracy of

traceability relations in case of fully automated approaches. We have detected that some tools do not provide efficient means for consistency and completeness checking. In addition, the absence of proper querying mechanisms hinders the analysis of traceability information. One other conclusion that can be drawn from the diversity of the existing tools and their approaches is that there is a need for standard and widely accepted software architecture traceability tools that would be available for researchers and practitioners. The absence of a proper support for completeness and consistency checking, and querying mechanisms, as well as standard and widely accepted tool support can therefore be identified as a major direction for future work.

With respect to the empirical evidence, ten of the reported traceability approaches between software architecture and the source code are empirically validated or evaluated in some form. Out of those ten approaches, four approaches are only evaluated in case studies. One approach is evaluated using action research. Four of the other approaches are validated in small experiments. Finally, one approach is evaluated using a retrospective study. Hence, we can conclude that the empirical evidence for architectural traceability is rather weak so far and that there is a strong need for more empirical evidence in the area of software architecture traceability. For example, there is a need to empirically evaluate the role of architecture traceability in understanding software architectures.

One of the most widely reported benefits of traceability approaches is that they help developers and maintainers to identify changes by determining the artefacts that are affected by change and thus estimating the effort for applying a particular change. It is also reported that they help in answering various comprehension or understandability related questions like: What is the role of a particular software architecture artefact in the code base? Which architectural or code base elements have a causal relationship to which project artefacts? In addition, traceability supports in verification and validation of software systems. For example, traceability approaches can be used to check whether an architecture is complete and consistent with the code base. This is one reason why various researchers have claimed that traceability plays an important role in better reasoning and understanding of software systems.

Despite numerous years of research and all the potential benefits of traceability, it is still not as widely adopted in industrial settings as it could be. We have recognized various liabilities, issues, and challenges. All traceability approaches require some additional effort and introduce an additional level of complexity to software development projects (but might help to resolve other complexities). Other reported liabilities of architectural traceability highly depend on the chosen traceability approach. For instance, fine-grained traceability is more precise than coarse-grained traceability, but developers might be overwhelmed by the number of traceability links that are created and hence miss the "big picture" aspects. The manual traceability approaches tends to be inconvenient, time consuming, complex, error-prone, and costly. In order to reduce manual traceability issues, various researchers have proposed semi-automatic or automatic traceability approaches. But unfortunately, the reported automated traceability approaches fail in producing accurate results and tend to produce results with low precision and proliferation

of traceability links that are difficult to manage and understand. Most of these approaches require human assistance in verifying the generated traces, in particular, identifying the false positives and missed traces. Some of these approaches even re-generate all traceability links upon modifications in development artefacts, which might lead to problems caused by overwriting manual changes. In addition, most of the existing traceability approaches between software architecture and source code provide only insufficient support to address various architectural concerns, for instance there is a need to more precisely address various stakeholder issues, quality concerns, and software development artefacts across various architecture or design views.

Based on our SLR of current software architecture traceability approaches, it can be assessed that semi-automatic traceability appear to be more well suited for many tasks compared to both manual and automatic traceability approaches. However, issues with manual and automated traceability approaches can be reduced through the following measures: Manual traceability approaches can be improved by setting up the links while working on the architecture or the design, as it is rather inconvenient and error prone for developers to establish numerous traceability links after finishing the design or architecting phase. In case of automated traceability approaches, minimalistic and non-redundant traceability links can be generated, by retaining only effective traceability links for a specific purpose such as supporting impact analysis. Automated approaches also require better support for restructuring of traceability links, because they appear to be inappropriate for establishing and maintaining traceability links at various levels of granularity and abstraction.

5. CONCLUSION AND FUTURE WORK

This study presents the first SLR in the area of traceability between software architecture and source code, resulting in the selection of 11 unique traceability approaches. Our results show a real need for stronger empirical evidence for software architecture traceability approaches, as well as proper support for completeness and consistency checking, querying mechanisms and standard and widely accepted tool-support. This article has presented a classification scheme to distinguish various aspects of traceability approaches, based on the nature of approaches, automation of traceability approaches, types of traceability relations, granularity of traceability approaches, traceability direction, and representation of traceability information. By offering this categorization, our SLR provides a foundation to guide researchers and practitioners to select or study a specific approach or set of approaches that is of interest to them.

Our future research goal is to build an empirically supported body of knowledge to investigate in how far explicit traceability links help in the understanding of architectural models and their links to other software development artefacts. To do this, we plan to perform a series of empirical experiments. In addition, we will further analyse the identified gaps and liabilities in the approaches especially regarding traceability in architectural views and to the code base.

6. ACKNOWLEDGEMENTS

7. REFERENCES

[1] J. Alves-Foss, D. Conte de Leon, and P. Oman. Experiments in the use of xml to enhance traceability between object-oriented design specifications and source code. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9 - Volume 9*, HICSS '02, pages 276–, Washington, DC, USA, 2002. IEEE Computer Society.

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

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Prentice Hall, 2003.

[4] G. Buchgeher and R. Weinreich. Automatic tracing of decisions to architecture and implementation. In *Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, WICSA '11, pages 46–55, Washington, DC, USA, 2011. IEEE Computer Society.

[5] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Softw. Pract. Exper.*, 41(4):363–392, Apr. 2011.

[6] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[8] T. Haitzer and U. Zdun. Dsl-based support for semi-automated architectural component model abstraction throughout the software lifecycle. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, QoSA '12, pages 61–70, New York, NY, USA, 2012. ACM.

[9] M. Hammad, M. L. Collard, and J. I. Maletic. Automatically identifying changes that impact code-to-design traceability during evolution. *Software Quality Control*, 19(1):35–64, Mar. 2011.

[10] B. Kitchenham and S. Charters. Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[11] B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman. Systematic literature reviews in software engineering - a systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, Jan. 2009.

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

[13] P. Mäder, O. Gotel, and I. Philippow. Enabling automated traceability maintenance through the upkeep of traceability relations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 174–189, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] E. Mamas and K. Kontogiannis. Towards portable source code representations using xml. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, WCRE '00, pages 172–, Washington, DC, USA, 2000. IEEE Computer Society.

[15] M. Mirakhorli and J. Cleland-Huang. Tracing architectural concerns in high assurance systems (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 908–911, New York, NY, USA, 2011. ACM.

[16] M. Mirakhorli and J. Cleland-Huang. Using tactic traceability information models to reduce the risk of architectural degradation during system maintenance. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 123–132, Washington, DC, USA, 2011. IEEE Computer Society.

[17] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 639–649, Piscataway, NJ, USA, 2012. IEEE Press.

[18] L. G. Murta, A. Hoek, and C. M. Werner. Continuous and automated evolution of architecture-to-implementation traceability links. *Automated Software Engg.*, 15(1):75–107, Mar. 2008.

[19] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner. Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 135–144, Washington, DC, USA, 2006. IEEE Computer Society.

[20] T. N. Nguyen, E. V. Munson, and J. T. Boyland. The molhado hypertext versioning system. In *Proceedings of the fifteenth ACM conference on Hypertext and hypermedia*, HYPERTEXT '04, pages 185–194, New York, NY, USA, 2004. ACM.

[21] T. N. Nguyen, E. V. Munson, and C. Thao. Object-oriented configuration management technology can improve software architectural traceability. In *Proceedings of the Third ACIS Int'l Conference on Software Engineering Research, Management and Applications*, SERA '05, pages 86–93, Washington, DC, USA, 2005. IEEE Computer Society.

[22] M. Riebisch. Supporting evolutionary development by feature models and traceability links. In *Proceedings of the 11th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS2004), Brno, Czech Republic*, ECBS '04, pages 370–377, Washington, DC, USA, 2004. IEEE Computer Society.

[23] H. Tran, U. Zdun, and S. Dustdar. Vbtrace: using view-based and model-driven development to support traceability in process-driven soas. *Softw. Syst. Model.*, 10(1):5–29, Feb. 2011.

[24] N. Ubayashi and Y. Kamei. Architectural point mapping for design traceability. In *Proceedings of the eleventh workshop on Foundations of Aspect-Oriented Languages*, FOAL '12, pages 39–44, New York, NY, USA, 2012. ACM.