A Survey-driven Feature Model for Software Traceability Approaches

Edouard Romari Batot, Sébastien Gérard, Jordi Cabot

A Survey-driven Feature Model for Software Traceability Approaches

Edouard Batot SOM-UOC ebatot@uoc.edu
Sebastien Gerard CEA LIST sebastien.gerard@cea.fr
Jordi Cabot ICREA & UOC jordi.cabot@icrea.cat

Abstract—Traceability is the capability to represent, understand and analyze the relationships between software artefacts. Traceability is at the core of many software engineering activities. This is a blessing in disguise as traceability research is scattered among various research subfields which impairs a global view and integration of the different innovations around the recording, identification and management of traces. This also limits the adoption of traceability solutions in industry.
In this sense, the goal of this paper is to present a characterization of the traceability mechanism as a feature model depicting the shared and variable elements in any traceability proposal. The features in the model are derived from a survey of papers related to traceability published in the literature. We believe this feature model is useful to assess and compare different proposals and provide a common terminology and background that could speed up the creation of new ones on top of them. Beyond the feature model, the survey we conducted also help us to identify a number of challenges to be solved in order to move traceability forward, especially in a context where, due to the increasing importance of AI techniques in Software Engineering, traces are more important than ever in order to be able to reproduce and explain AI decisions.

Index Terms—Software Engineering, Model-Driven Development, Traceability, Feature Model, Explainability

1 Introduction

The need for traceability has always been salient in software and systems development. Across the years, there has been a continuous interest in developing techniques to facilitate the representation and analysis of traces and links between related artefacts. It helps explaining their execution and evolution as required in many software engineering activities and disciplines such as code-generation, program understanding, software maintenance, and debugging.

The importance of traceability was first recognized in system engineering especially related to the development and certification of critical systems where it is a primary concern. As an example, traceability is part of any certification mechanism in all commercial software-based aerospace systems as stated in documents like the RTCA DO-178C (2012) [79, 65]. The consideration of various levels of abstraction in software development and the meaning of verification in model-based development paradigm - which figures abstract representations (models) as the core artefact for conceptualization - was latter introduced with companion documents (specifically, DO-331). The automotive industry has followed the same path with the construction of an international standard for functional safety [49].

Despite these important evidences on the need for explicit (and automated) tracing abilities in software development, traceability is not widely adopted, even less automated, and there is little feedback from its concrete use in industry [78] beyond the critical domains above.

There is a lack of global techniques to ease the manipulation of traces and automate tracing processes. Thereby, traceability in the industry, when required, ends up being mostly a manual process [60]. Moreover, with no standard definition or representation of traces, it is difficult to bridge the gaps between the different partial traceability solutions existing in research subfields [5, 106, 105]. Even the software engineering body of knowledge does not seem to properly consider the key relevance of traceability in software engineering as it only mentions traceability once [16].

Approaches vary greatly in their means and goals. The foundation for an effective modelling of traceability is disseminated among a profuse literature. Moreover, most approaches focus on specific pairs of artefacts and therefore remain difficult to integrate in industrial scenarios. This happens in a context where artificial intelligence techniques are being integrated in development processes, raising the need for more powerful reproducibility and explainability concerns, both requiring the assistance of traceability mechanisms.

This paper aims to provide a comprehensive perspective on the state of the art of traceability techniques in model-based software development and their limitations with the short-term goal of facilitating the evaluation and comparison of current solutions. And with the mid-term goal of accelerating the development of new traceability solutions that could benefit from the existing ones thanks to our new conceptualization in the form of a feature model describing the potential dimensions and concerns a traceability solution may wish to consider. We do not create the feature model or just based on our (partial) knowledge and expertise in the domain. Instead, we ground our classification with a survey of the published literature in this field. According to this survey, we group the traceability features in three main dimensions: trace definition, trace identification and trace management, with the corresponding feature hierarchies for each of them.

The paper is organized as follows. After a brief introduction, we discuss in Section 2 how our work compares to other meta studies and characterizations of traceability research. We then introduce some basic traceability terminology in Section 3. Section 4

describes how we conducted our literature review and Section 5 presents a detailed feature model derived from the survey of the retrieved works. This analysis also helps us to propose a number of discussion points and open challenges in Section 6 before concluding this work.

2 State of the art

Traceability was proposed, from the very beginning of software engineering, as a measure to ensure that a system being developed actually reflects its design. Already in the original NATO working conference, quality projects were praised for making "the system that they are designing contain explicit *traces* of the design process" [85]. From that point on, traceability has been studied from a myriad of perspectives, dimensions and applications.

As such, it is no surprise that there have been other previous attempts to characterize and summarize the state of the art in the traceability field. In what follows, we compare our own proposal with previous surveys of traceability papers and related work aiming to systematize what we know about traceability. As we will see, ours stands out by combining both types of works, i.e. by proposing a traceability systematic description grounded on a thorough analysis of traceability proposals in the literature, instead of offering a more descriptive survey or an individual and/or partial traceability model.

Publications around traceability started to grow in the 90's with a seminal work from Gotel *et al.* [36] with, probably, the first systematic analysis of the traceability problem. Lindval *et al.* corroborate their findings and endorse the usefulness of traceability for object/model oriented software development [58]. Since then, many researchers attempted to draft general traceability frameworks and methods. For instance, in 2007, Cleland Huang *et al.* described best practices that remain essential today [21]. They distinguish three categories of concern: the purpose and constraints of tracing in a specific environment; the creation of traceable artefacts with a project glossary, quality requirements and rich, organized content; and the automation of tracing processes. As we will see in the next section, these concerns are an important part of the feature model.

With the proliferation of traceability purposes, some authors explicitly asked for better sharing of experiences in using traceability [37] and evaluating the solutions existing so far [95]. Surveys and literature reviews trying to group and compare them began to appear as well, though most of them focused on specific subareas such as requirement engineering [36, 15], model-driven development [33, 105, 73, 90, 66], software product lines [100, 4], benchmarking [95], and information retrieval [24, 13, 40]. To complement this more scientific surveys, Konigs *et al.* survey industrial application of traceability approaches [55], showing its limited penetration. Neumuller *et al.* show that the adoption is worse in small businesses where traceability is even less automated [70]. Finally, Charalampidou *et al.* review traceability approaches in the prism of their empirical evaluation. Authors add to the conclusion of other surveys that "although many studies include some empirical validation", there is still much to be done with respect to validation and reproducibility [20].

These surveys point to some shared concerns, like the crucial lack of a common terminology. They mention that existing traceability solutions struggle to achieve satisfactory cost/benefit ratios in part because of the nonexistence of such common traceability knowledge base that facilitates the reusability and improvement of available traceability tools and techniques. This is aggravated by the fact that, as pointed out above, many of the proposals belong to different research subfields, which limits the discovery and awareness of alternative solutions. For instance, Winkler *et al.* point out that researchers in requirement engineering and in model-based development do not communicate enough among each others [105]. This lack of communication and shared understanding is one of the open challenges in the traceability domain [22, 5].

To solve this issue several works aim at proposing specific traceability models. Unfortunately, many investigations suffer a lack of generalizability due the specific nature of the problem being solved (*e.g.,* certification conformity [54], model transformation coevolution [39]), or the specific nature of the solution considered (*e.g.,* w.r.t. its language: SysML [68], w.r.t. its engineering field: SPL [4]).

As an example, the automatic identification of trace links is one of the most studied features. There are plenty of proposals to achieve this but as they are evaluated using different datasets and configurations, they cannot be directly compared [93, 40, 13]. Another example would be model-driven engineering, where the proposal and usage of traceability languages and models should be more "natural". Nevertheless, not even there we find a unified traceability representation model: Mustafa *et al.* argue that "the main issues in traceability nowadays are building traceability models that can accommodate the capturing of traceability information and providing common semantics for trace links" [66]. Proposals tend to focus also on a specific model-driven engineering problem: the co-evolution of models and transformations [3, 89, 74, 30] instead of aiming for more general solutions.

As a result of this confusing situation, a few authors asked for more standardized practices. These proposals are however restricted to specific application or engineering domains and miss their general target. Debiasi *et al.* propose to build a common body of knowledge on traceability. They refer to requirements traceability and focus on the organizational challenges of the implementation of traceability approaches [25]. Heisig *et al.* [44] present Capra, an Ecore implementation of a framework for the traceability of software product lines.

We agree with these authors that this lack of *de juro / de facto* traceability standard is hampering the benefits of current traceability solutions and hindering evolution in the field. This paper intends to cover this gap by proposing a traceability characterization that stems from the analysis of all existing proposals. We believe this model can be useful to researchers trying to improve traceability techniques in any subfield and to practitioners looking for a way to compare and choose the traceability solution that best suits their needs.

In the model-driven engineering community the use traceability specific languages together with automated model transformation appears as an ideal soil to grow end-to-end traceability. This led authors to present classifications and terminologies for a systematic perspective on the tracing of MDE development [73, 30, 89]. Unfortunately, this body of work mostly limits its focus to the maintenance and coevolution of model transformations. Holtmann *et al.* built a common terminology for the traceability of model-based systems [46]. We partly reuse it in the next section to extend the terminology to this specific field.

3 Towards a common traceablility terminology

A clear conclusion of the previous section is the lack of a common agreed upon conceptualization for traceability that helps evaluating, comparing and reusing traceability solutions over a variety of scenarios and application domains. Thus, the *incoherency problem* still arises in traceability research [104]. Even if an individual article makes a claim that withstood rigorous testing and statistical analysis, it might not use the same words as an adjacent article, or it would use the same words but intend different meanings. For instance, the term *traceability* is used to designate both the ability to trace system elements, and the traceability links (the relations) themselves [15, 5].

Therefore, before proposing our global traceability model, we first recap the different usages of the key traceability concepts and propose a unified definition that we will use in the rest of the paper.

3.1 Traceability components

Traceability research refers mainly to a definition from Gotel *et al.* that defines traceability as the ability to describe and follow the life-cycle of a requirement, from its initial specification to the design and code elements of the system implementing it [36]. This is still the most popular meaning for traceability [15, 9] even if modeling approaches try to generalize this notion by seeing traceability as a valuable tool to link all types of linking artefacts at either the same or different levels of abstraction [59, 99].

Regardless of the specific interpretation of traceability we observe a division of knowledge into four main areas:

- **Strategizing traceability**. It involves defining the explicit traceability purpose for the project at hand and how to best reach that goal.
- **Trace and artefact representation**. It covers the design / adaptation of a language to be used to define the traces and decisions regarding its syntax, expressiveness, variability, integrations, etc. For instance, this can be done by means of creating a full traceability domain-specific language.
- **Trace link identification**. It designates the identification of traces in a software system, be it a post-requirement assisted elicitation, a live record during a system execution or an automatic AI-based inference process. This latter approach is the clear trend right now to help the identification of links between heterogeneous artefacts.
- **Trace management**. It refers to the ways to use and maintain the traces. This includes tool support for the persistence, retrieval, and analysis of traces.

The first area is a high-level concern that influences the requirements of the other three to cover the specific needs of a project. These three will therefore be used to structure our feature model later on. Note that the representation component should be part of any traceability solution as it is the base component to be able to, at the very least, express traceability information.

3.2 Traceability glossary

We propose some general definitions for the most frequently encountered traceability terms while searching for and studying solutions for traceability in any of the above categories. These definitions, partly borrowed from past literature [37][hotlmann2020-MB-traceability-terminology], aim to encompass the different uses and dimensions of traceability depicted above. Our set of terms is not exhaustive but provide a common core generic enough to be then adapted to specific scenarios. This is also why we try to be precise with the definitions while also offering room for slightly different (but compatible) interpretations.

- **Traceability** is the ability to trace different artefacts of a system (of systems). It is defined in the IEEE Standard Glossary of Software Engineering Terminology [48] as

  1) The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor–successor or master–subordinate relationship to one another. [...]
  2) The degree to which each element in a software development product establishes its reason for existing.

  Gotel *et al.* define traceability as "requirements traceability [which] refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction" [36]. Aizenbud-Reshef and colleagues extend the Gotel's definition of traceability and define MDE traceability as "any relationship that exists between artifacts involved in the software engineering life cycle" [2].
- **End-to-end traceability** refers to a complete and ubiquitous traceability application, comprising a set of traces that extend throughout the entire life of a development project, from the requirements phase to, test, exploitation and retirement phases. "End-to-end traceability weaves artifacts together in tandem with the various phases of the life cycle" [7].
- A **trace** is a path from one artefact to another. A trace is composed of atomic **links** that directly relate artefacts with each others. The representation of traces, their data structure and behaviour, is defined in a traceability grammar or metamodel [27] depending on how the trace language is defined. In any case, the language definition specifies the concepts and relationships available to define traces. As discussed before, no standard language has emerged yet.
- An **artefact** can be any element of a system - *e.g.,* unstructured documentation, source code, design diagrams, test cases and suites... The nature of artefacts follows two main dimensions: the life cycle phase they belong to (*e.g.,* specification, design, implementation, test), and their type (*e.g.,* unstructured natural language, grammar-based code, model-based artefact). The **granularity of artefacts** is the level to which artefacts can be decomposed into sub parts. We call a **fragment**, the resulting product of the decomposition of an artefact. A fragment can be itself broken down into smaller parts (or sub-fragments), and so on.
- A **link** is a direct relationship between two artefacts. Links can be typed to better support the heterogeneous nature of traceability applications. The type of the link can help express the rationale behind the relationship - it informs not only *how* artefacts are linked but also *why* [60]. Typing is a primary concern in conceptual modeling in general [71]. This link definition is consistent with the concept of link in popular modeling languages like UML or SysML. *Link* is a specialization of the concept of Dependence (which

is itself a specialization of Relationship) which is used to explicitly model a traceability relation between two sets of elements. We add the need of additional typing to this relationship.
Links can be explicit or implicit. An **implicit link** shows artefacts bondage at a syntactic or semantic level without the need for an **explicit link** to be part of the model (*e.g.,* a binary class and its respective source code artefact are implicitly "linked" to each other, yet this bondage is not part of any language or grammar definition) [73].

- An **agent** is the (human) actor accountable for an artefact, or a link.
- **Application and engineering traceability domains:** the specific nature of a traceability project follows two dimensions: i) the domain of the target - that is, the application domain, and ii) the domain of solution considered - the engineering domain.
- **Trace integrity** is the degree of reliability that bares a trace. It is an indirect measure that includes, for example, both the age of a trace, the volatility of artefacts targeted by the trace, and the automation level of tracing features. This indication is supported by **evidences** that can be quantitative or qualitative. For example, how long (how many versions ago) has the trace been identified in the system? Or, has the trace been identified manually or automatically? Is there an automated co-evolution mechanism between traces and targeted artefacts? What is the level of experience of the trustee who identified it? The volatility of source and target artefacts are also factors that may influence the relevance and accuracy of a trace.
- **Pre-requirement and post-requirement** traceability refer to, respectively, traces identified during specifications elicitation and during the implementation (design and code) step of a specification [36]. The IEEE Guide for Software Requirements Specifications mentions **forward** and **backward** traceability, referring to the ability to follow traceability links from a source to a specific artefact, or the opposite from the artefact to its source respectively [47] but, technically, the direction of traceability link (from source to target, or from target to source) does not make a difference.
- **Vertical traceability** refers to the linkage between artefacts at different levels of abstraction (*e.g.,* derives, implements, inherits) whereas **horizontal traceability** refers to artefacts at the same level (*e.g.,* uses, depends on).
Vertical and horizontal traceability are defined in the opposite way in the literature on impact analysis where "vertical traceability refers to the ability to trace dependent artefacts within a model, while horizontal traceability refers to the ability to trace artefacts between different models" [82, 58, 23].
We chose the first definition since it seems more fit to a general idea of software systems, rather than the peculiar scope of impact analysis.
- **Time related traceability** goes along two dimensions: the evolution of (a group of) elements through successive development tasks, or the evolution of artefact properties during an execution of the system.

On top of these concepts, a recent work, by Holtmann *et al.*, makes a distinction between a *foundational* and a *specifically model-based* terminology [46]. This latter add a specification for *model* and *language* scope definitions, as well as a distinction between *relational* and *referential* trace links.

- **Intra/Inter model** trace links differentiate between relations that links elements of the same instance of the language and relations linking elements from distinct instances. This distinction was first introduced by Lindval *et al.* [58].
- **Intra/Inter DSL** differentiate between relations that links elements in models based on the same language and relations that links elements in models from different languages.
- The distinction between **Relational and Referential** trace links lies in the instantiation (or not) of the instance link. "A relational trace link is represented by *a dedicated node* with incident directed edges pointing to the trace artifact nodes" whereas "a referential trace link is *a directed edge* from one trace artifact node to another trace artifact node". In the latter case, a trace link is commonly represented as a *property* of the source artefact.

Some of these concepts will explicitly appear in our feature traceability model while others act as requirements and usages that should be supported/facilitated by the features in the model and taken into account when choosing a specific traceability solution depending on how well that solution covers the specific features of interest for the project at hand.

4 Traceability Survey Method

In this section we depict the methodology we followed to collect papers proposing traceability solutions, including at the very least the core *representation* component (see previous section). The analysis of these papers will give rise to the feature model we will present next.

The selection process combined the manual selection of a few approaches based on our own experience working in this field and on the works covered by other meta-studies [37, 5, 22, 40] together with a systematic literature search by mining bibliographic data sources following the literature review process established by Kitchenham and Charters [52]. Fig. 1 depicts the three main steps of the process.

4.1 Data source and search strategy

We used DBLP (2020-07-01 [1]) as our core electronic database to search for primary studies on traceability. To avoid missing possibly relevant approaches, we decided not to put a specific period constraint for the search, but we limited the scope of the search to paper of five pages or more to avoid opinion and vision papers, posters, tool demos and other types of short papers to reduce the number of results while maximizing their quality.

Based on the topic of this survey, we defined the terms of the search query according to the recommendations of Kitchenham and Charters [52]. We apply the query on the title and abstract of potential relevant publications. As using very generic terms like “trace” or “traceability” returned thousands of results, we decided to combine in the search query trace-related keywords with language-related ones since we target traceability proposals that discuss how traces need to be represented. We are not necessarily interested in all publications figuring concrete applications of traceability but rather modelling (in a broad sens)

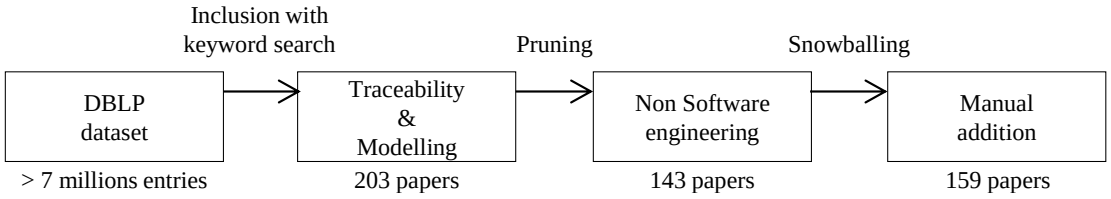

Fig. 1: Survey Process.

work on traceability. We narrowed the scrutiny of the search to work on traceability (modelling tracing) only and we refined the query including keywords more akin to modeling and software engineering. As many traceability languages are model-based, as part of the language variations we included model, modeling and other core MDE concepts. This brought down the results to 203 papers.

Here is the exact query we applied: `.*(([Tt]rac(eability|ing))|([Tt]race[rs])).* AND .*(([Mm]odel[- ])(([Dd]riven)|([Bb]ased))|MD[DAE]|Model[l]ing|[Tt]ransformation|DSL|[Ll]anguage).*`

4.2 Pruning

In what follows we describe our inclusion and exclusion criteria. We further explain how we applied these criteria on the previous set of papers.

Inclusion criteria

1) the paper is a technical contribution
2) the paper is about tracing in software engineering
3) traceability is the main concern of the paper

Exclusion criteria

1) the paper is not a primary study
2) the paper is not a white paper

Before we applied these criteria on the potential papers fetched by our query, we removed automatically papers of less than 5 pages long. We also automatically extracted papers whose titles mentioned "biology", "education", "kinetics", "logistics", "physiology", "physics", "neuroscience", "agriculture", and "food" which appeared each in a couple of results. We manually examined the 183 papers left and excluded 40 papers that did not fulfilled the criteria or were duplicates.

4.3 Snowballing

At the end of the previous steps, we double-checked that we did not miss any potentially relevant approach due to a number of reasons, *e.g.*, some workshop papers are only indexed by ACM or papers that may be using different synonyms to traceability like “composition” or “extension”.

Finally, we added papers we were aware of (if not already in the result set) and a few more we found by snowballing on the selected papers references. They amount to a total of 10 papers. This lead to a final result of 159 papers. Among them, there are 41 journal articles, 82 in conference proceedings, and 36 workshop reports (see Table 1). Fig. 2 shows the chronological distribution of the selected publications.

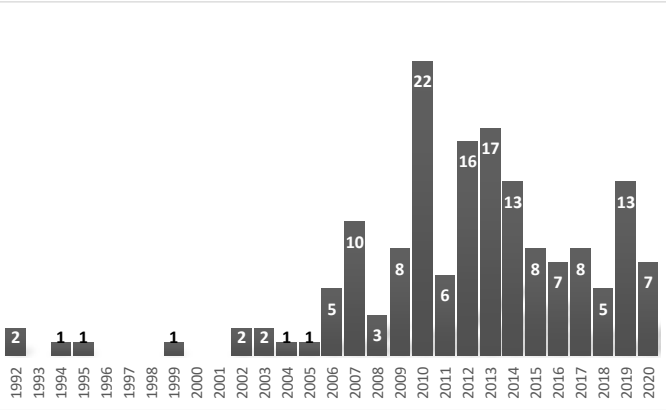

Fig. 2: Papers selected related to traceability and modeling.

Publication type	
Journal	41
Conference	82
Workshop	36

TABLE 1: Publication types of the selected papers.

4.4 Threats to validity in the selection process

We acknowledge limitations in the execution of our survey method. First, we only used DBLP as a source database. Yet, it is recognized as a representative electronic database for scientific publications on software engineering and already contains more than five million publications from more than two million authors. Setting the limit based on the number of pages alone to elude short papers is another threat to validity. Yet, it is a reproducible practice that limits the number of papers to analyse and thus helps concentrate on the topic rather than the engineering of the survey. Then, the vocabulary related to traceability is scattered among various fields of application with their respective nuances. We mitigate the risk of missing papers by manually adding papers that were not using variations of this term but were still referenced by papers that did. Still, focusing on traceability as a key term was also a conscious decision as we wanted to characterize the works in this field, focusing on those papers that define themselves as part of it. Finally, we acknowledge that many other works will not include the "representation" component, yet they could help eliciting features for the outcome of this study, but we do not target to define a precise traceability information model for specific uses but rather a holistic view on the field that do not lose itself in too many (unnecessary) details.

5 A FEATURE MODEL TO CHARACTERIZE SOFTWARE TRACEABILITY

This section presents our feature model describing the traceability features and dimensions found in the analysis of the literature conducted in the previous section. Our feature model groups them by similarity and provides additional descriptions on the most important aspects of each one, *e.g.*, different existing alternative implementation of the same feature and/or the most/the least studied ones in each group.

Next subsections provide some background on feature modeling and then zoom in each of the three main dimensions of traceability: trace representation, trace identification, and trace management. These dimensions are depicted in Fig. 3, Fig. 4, and Fig. 5, respectively.

5.1 Introduction to feature modelling

A feature model leverages features as the abstraction mechanism to reason about product variability. It is a hierarchically arranged set of features, where relationships between a parent feature and its child features may be categorized as: *and* – all sub-features must be selected, *alternative* - only one subfeature can be selected, *inclusive or* – one or more can be selected, *mandatory* - features that are required, and *optional* - features that are optional [51]. Each feature represents an increment in product functionality.

Feature modeling is a technique that has been intensively used for documenting the points of variability in a software product line, how the points of variability constraint one another, and what constitutes a complete configuration of the system. But beyond product lines, feature model are also more and more used to shed light on complex domains by representing the core concerns and variation points in a complex ecosystems (*e.g.*, [17]), as we do in this paper.

5.2 Trace definition and representation

All approaches must discuss their representation of trace artefacts even if they can differ on the type of traces they consider and the application they target. Representations are so diverse that our survey selected more than 80 papers mentioning their own distinct definition for traceability – with 20 metamodels effectively depicted in those papers. Some researchers present generic graph-based representations [91, 38] while others focus on representations much more specific to a concrete application like this metamodel for change impact analysis [35] or multi-model consistency [98]. In both cases, what traceability approaches target and how they represent a trace is differently approached.

Fig. 3 shows the hierarchy of features related to the definition and the representation of trace artefacts. A peculiar focus is put on the typing of traces' relationships. Typing relationships is important to add semantics to the trace so that the engineer can know not only what are the linked artefacts but also why they are linked. As such, it facilitates the application of traceability solutions to specific domains. We also detail the genericity of the language, the nature of the artefacts covered by the traceability proposal, and the possibility to annotate traces with quality properties.

We would like to remark the contribution of model-based approaches for traceability in this subsection. The use of MDE tooling such as ATL [88, 50], or the Eclipse Modeling Framework (EMF) allows the automated generation of traceability information as a side effect of executing operations [33, 105]. The modeling community has proposed metamodels for end-to-end traceability [44, 42], as well as metamodels specific to engineering domain such as model transformation [50, 4, 101, 11] or software product line [50, 101]. Paige *et al.* call for more flexible modeling where models of different formats are associated to each others' with annotations that allow automated bond or dependency inference between both application and engineering domains [93, 75].

5.2.1 Language

Languages specific to traceability provide the ability to represent trace artefacts with increased relevance and accuracy. Yet, they often suffer the limitation to be built *ad hoc* and lack a significant power of reusability into other domains and risk of ending up reinventing the wheel. Among these domain-specific languages for traceability, some authors attempt a generic definition of traceability [44, 8] while others provide a language specific to a single domain, *e.g.*, traceability for software product lines [4].

We found few studies interested in the use of general-purpose software language for traceability - even though this would be appealing to industrial partners interested in instrumenting their legacy systems code with traceability information to facilitate future evolution or migrations [68]. Another type of general languages for traceability could involve representing traces in spreadsheets, text files, or databases. This shows better learning curves than using a domain specific language at the cost of a cognitive gap between software engineers and domain experts. As an unfortunate consequence, "the maintenance costs turns out to grow accordingly [to the usability of generic representations] and team members fail to keep the trace artefacts up-to-date" [21].

A potential sweet spot could be to "plug" traceability concerns on top of other languages like SysML [68] to benefit from an existing language structure while keeping most of the benefits of using a DSL.

5.2.2 Artefacts targeted

In relation to the artefacts targeted by traceability purposes we distinguish between the nature of the artefact and its granularity as both dimensions are important and used in the literature.

For the nature aspect, on the one hand, investigations differ on the development phase they target. Linking requirement specifications to design and code level predominate in the literature with more than 50% of the papers in the survey addressing requirement traceability. Other phases such as test and verification are targeted as well but in a lesser proportion. On the other hand, the type of the artefacts is important to deduce the level of potential generalization to other phases of the software lifecycle. Papers focus on four different types: unstructured document, structured as grammar-, and model-based artefacts, and binaries.

With regard to the granularity of the artefacts targeted, *i.e.*, their level of decomposition, some approaches go for a customizable granularity to adapt to artefact hierarchies while others focus on specific types of artefacts (*e.g.,* to concentrate their work on specific optimizations of trace identification).

5.2.3 Relationship types

As many authors have demonstrated, offering the ability to the user to define personalized types of relations between the artefacts of a system fosters the comprehensibility of the traces produced [71]. We distinguish between approaches offering predefined types and approaches allowing custom typing. Often the predefined types relate to the field of software engineering (implements,

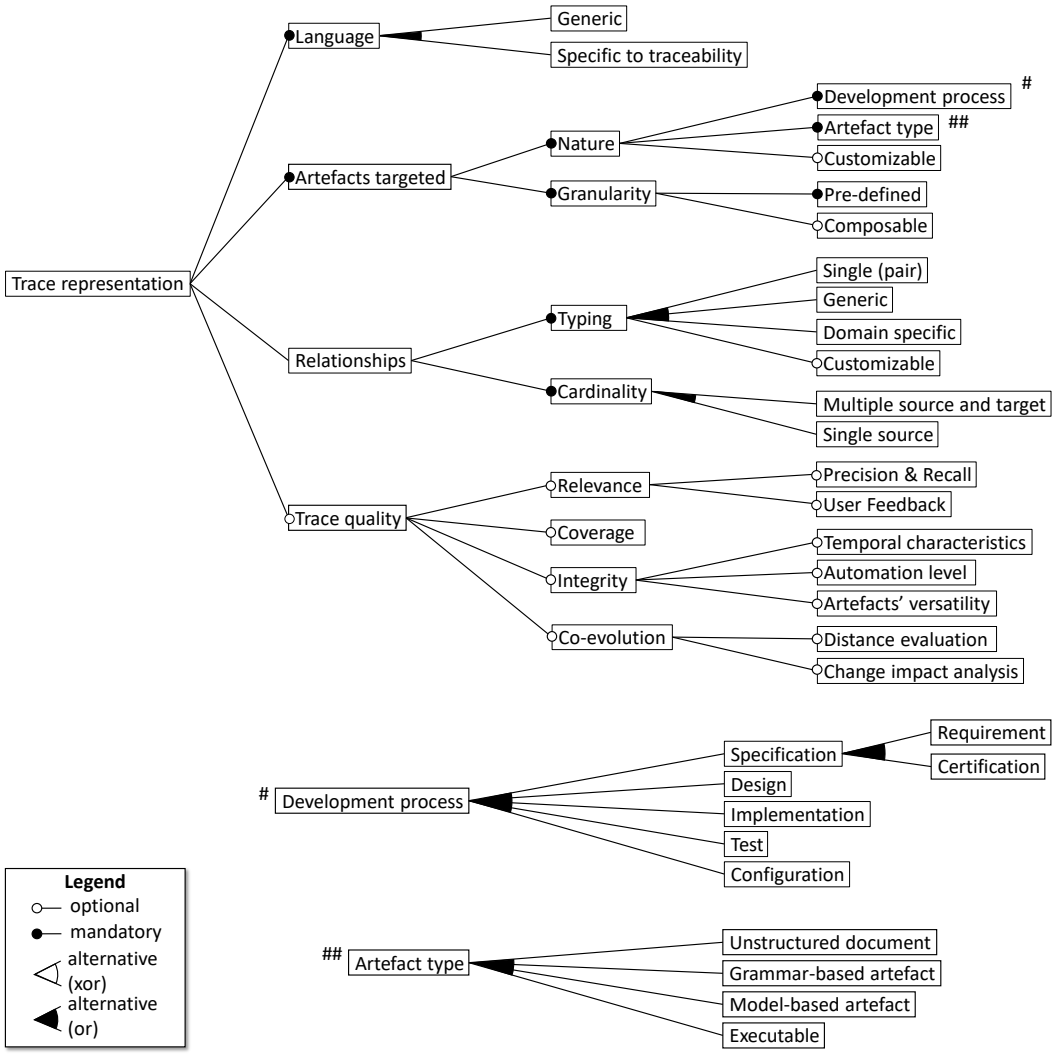

Fig. 3: Features related to the representation of a trace.

inherits, uses, executes ...), but not only. For example, Maletic *et al.* mention that a separation between *causal*, *non causal*, and *navigation* relationships can be appropriate [61]. Predefined types allow increased monitoring and user-friendliness to developers. They are found in most contributions relating the optimization of trace identification.

Obviously a fixed typing facilitates the analysis of the traces as the potential set of semantics and interpretations are fixed while offering domain-specific types increases the usability and comprehensibility of the approach. As an example, SysML v2 is offering a more powerful mechanism to define links between artefacts compared to the previous SysML version (where we had a sole dependency-like mechanism).

Allowing users to define the types of relationships specific to their area of expertise helps to fill the gap between the design and the use of tracing functionalities [106]

The literature shows also a distinction between approaches considering relationships with multiple sources and targets and relationships allowing only a single source.

5.2.4 *Trace quality*

In most of the papers we studied, quality aspects were barely mentioned. It seems quality of the generated traces is not a major concern, or at least storing and annotating the traces with such information is not.

Yet, a few studies mention coverage and integrity. The coverage of a set of execution traces is used in approaches for software testing [34]. Coverage is also used by Rath *et al.* who address the problem of missing links between commits and issues with a classifier they train on textual commit information to identify missing links between issues and commits (*i.e.,* a lack in the coverage indicates such missing links) [86]. Matrix-based visualizations are particularly fit to assist coverage related tasks (See Section 5.4.1. Integrity of traces is addressed in work on model transformation where co-evolution figures an automatic verification of their coherence with other (versatile) software artefacts [98, 96]. In the same manner, Heisig *et al.* tag links which ends artefacts have been modified or deleted to inform the user of such changes [44]. The co-evolution of traces implies measuring distances between artefacts (syntactic, cognitive, geographic, cultural...) [10]. It also refers to the analysis of the changes of the system that impact traceability artefacts [35, 102]. In our survey, nine papers address artefacts co-evolution and 17 tackle model transformation limitations. These latter are a valuable tool to automate co-evolution tasks. In the many studies focusing on the optimization of link identification, the quality of the

results is mainly evaluated with precision and recall measurements and never rely on inherent trace artefacts characteristics. Few researchers include a user feedback [13].

A few publications relate the quality of their work to the computation of aggregated values, evaluated against company (or project specific) thresholds [19]. They make use of rules to automate the computation of customizable analyses and show that query, metric and rules are a powerful combination to measure the productivity of new initiatives.

5.3 Trace identification

Fig. 4 shows the hierarchy of features related to the identification of traces with four main possible categories: the manual elicitation of traces, their live record during execution and evolution, rule-based alternatives to assist the user with automation potential, and AI-augmented identification with domain contextualization.

5.3.1 Manual elicitation

Manual elicitation makes possible to create traces in an *ad hoc* manner. As an example, one of our industrial partner chose to hire a developer to elicit trace links necessary for a certification commitment. This was chosen rather than a (semi-)automated approach as they were not convinced the effort of augmenting an existing tool would pay off for that specific project.

5.3.2 Recording instrumentation

Teams can instrument the live record of traces during the execution and the evolution of software artefacts. This way traces recording the system changes are a side-effect of those same changes. There are initiatives to instrument existing languages such as ATL with rich log generation [88, 56], while others consider trace record an aspect that can be weaved with current existing languages [81, 88]. Ziegenhagen *et al.* mix execution traces with metadatas [108], and use developer interaction records [107] to enrich existing traceability artefact.

Model transformations are considered the hearth and soul of software modeling and, consequently, numerous studies attempt to enrich trace generation during transformation execution [101, 87, 56]. This ubiquitous integration (see Fig. 5, bottom branch) allows a semantically rich tracing of target and source artefacts [74]. Unfortunately, this option can only be applied when the system is being built, not when the system is already in place.

5.3.3 Identification rules

Once a system is in place, teams can identify rules that help retrieve and maintain traceability relations [67, 97]. Nentwich *et al.* describe a novel semantics for first-order logic that produces links instead of truth values and give an account of our content management strategy that provides rule-based link generation and consistency check [69]. At the model level, Grammel *et al.* use a graph-based model matching technique to exploit metamodel matching techniques for the generation of trace links for arbitrary source and target models [38], and Saada *et al.* recover execution traces of model transformation using genetic algorithms [87].

5.3.4 Domain contextualisation

Back in 1992, Borillo *et al.* published an article on the use of information retrieval techniques for linguistics applied to spatial software engineering [14]. This precursor work opened the box for AI-augmented traceability where machine learning algorithms help extract knowledge specific to the application domain (later called domain-contextualized traceability [41]). This is specially useful when the source (or target) of the trace link is an unstructured document or when such document is key to infer traces among other artefacts.

Researchers first extracted word vectors from natural language. Vectors intend to take account of the neighbouring words a term may relate to in the application domain [24]. This effort made the identification of bonds between requirement specifications and other artefacts possible with a gradually improving precision. Since then, many other information retrieval techniques for natural language processing were applied with success [6]. Studies on domain contextualization are separated into three subgroups according to the type of tools used (algebraic information retrieval models, statistical language models, and neural networks). For example, Florez *et al.* derive fine grained requirement to source code links [32], Rath *et al.* complete missing links between commits and issues [86], Marcus *et al.* identify links between documentation and source code [63]. An interesting publication from Poshyvanyk *et al.* shows that mixing expertise both in information retrieval techniques and engineering domains gives far better results than when taken separately [83]. McMillan *et al.* add that using structural information together with textual information benefits automated link recovery [64]. They combine information from both sources into a middle ground representation based on traceability link graph that "encodes all recovered textual and structural links as edges between nodes representing either program requirements or source code methods". We do not discuss in this paper the techniques related to data collection and training optimization. These are important features for automated learning which are discussed in depth in specialized literature.

Today, domain contextualization by means of machine learning for topic modeling, word embedding, and more generally knowledge extraction from unorganized text documents is the most popular traceability feature [40, 106]. We found 22 approaches dedicated to this topic alone in our survey.

Teams are also using genetic algorithms here, not to recover traces themselves but to cope with the variety of algorithms and parameters these approaches use [62, 76], and structural information to foster methodologies interweaving [77]. Unfortunately, a common critique rose against these positive results. Too many teams compete with each others to accomplish a better precision and recall when there is no standard to the effective quantification of traces artefacts into such variables. Too few attempt at qualifying the overall relation between these measurement and the effective impact on software development [22].

In that regard, Shin *et al.* propose a set of guidelines for benchmarking automated traceability techniques. Their evaluation (of 24 approaches) shows that methods of evaluation (when they are used appropriately) sometimes are not suitable to other application domains and that the variation in evaluation results across project is not investigated [95]. This corroborate Borg *et al.* who, in a systematic literature mapping on information retrieval approaches to traceability, notice that there are no empirical evidence that any IR model outperforms another model consistently [13]. The ability to continuously improve the learning process is mentioned in the literature but we found no evidence of its application.

5.3.5 Tool assessment

Borg *et al.* published a taxonomy for information retrieval techniques applied to traceability [12]. They emphasize with great

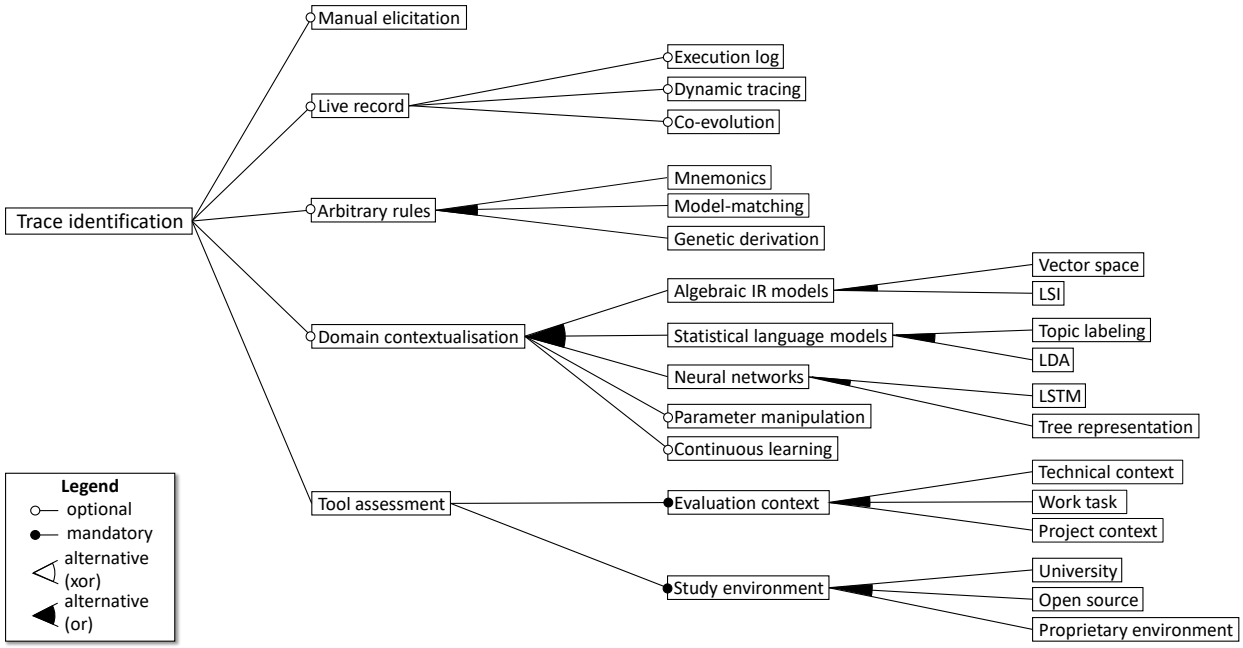

Fig. 4: Features related to the identification of trace links

concern the importance of the assessment of the tooling used to derive or identify traces. More specifically, the authors draw a differentiation between two orthogonal dimensions: the evaluation context that precises *where* in the context the tool is assessed (*e.g.,* at a technical, work task, or project level); and, the study environment that shows the kind of data used to fulfil the assessment (*e.g.,* proprietary, open source, or academic). These features will affect the measurable attributes used for the assessment as well as their generalizability.

A lack of industrial case studies together with a very strong trend to report solely for precision and recall values. This indicates an important issue in the automated identification of traces and may justify the weak investment of industry in this sector [13, 72].

5.4 Trace management

Fig. 5 shows the hierarchy of features related to the management of trace artefacts. We distinguish between the actual maintenance of trace artefacts, the evaluation of their integrity, their means of persistence, and their level of integration in running software systems.

5.4.1 Trace Maintenance

Trace links may be affected by changes on the artefacts they link (directly or transitively) and therefore can easily become obsolete. This gradual decay must be seriously taken into account to avoid having to re-elicit traces every time they need to be analyzed. A manual maintenance is not always impossible but not typically feasible in practice due to the amount of information such inspections would involve. Co-evolution techniques [67, 28, 84] attempt to tackle the burden to maintain trace links up-to-date [92, 19].

Beyond being able to manipulate traces, we also need to offer proper ways to visualize and inspect them [31]. The use of graphical representations stimulate human perception and the integration of such technique in traceability frameworks is a useful feature to augment user awareness [44]. On the other hand, matrix-based views offer a valuable perspective to understand and analyse traces [57]. They are particularly efficient in assisting the visualization of coverage related characteristics of traceability [34, 86].

In parallel, allowing a rich formulation of queries to assist the exploration of existing traces will help with reducing the amount of information users need to navigate through [19]. More precisely, structured text, in the form of metamodel instances or XML sheets allows query-based mining of trace datasets [26]. Interaction wise, hyper-text links is a *de facto* standard to browse trace links. Indeed, following links through successive clicks has become almost natural. Querying relies on the type of representation of traceability artefacts: SQL-like languages benefit from a long history of information mining while dedicated languages offers better legibility. Genetic programming has also permitted the automation of query formulation [80].

5.4.2 Trace Integrity

To cope with the decay and volatility mentioned above, ways to determine the integrity of existing traces is greatly needed. Work on these questions, although called out loudly by literature studies, is scarce in practice [105, 5]. The first option is given with manual annotation or vetting of trace links to inform about their level of reliability. Annotations allow a qualitative and quantitative evaluation [18]. This is the case for back-propagation of verification and validation results between design and requirements [43].

Some approaches enable the definition of invariant rules while manipulating traces or their targets [19]. If the invariant is violated, an exception for that trace is automatically generated. For example, we could define a rule that is violated when a change occurs in an artefact targeted by a trace if the corresponding link was identified more than two versions prior to the current version. In the same vein, Heisig *et al.* tag trace links when their target (or source) artefacts are modified or deleted [44]. Thanks to the ubiquitous integration of the tool, a warning is raised consequently in the Eclipse Modelling Framework.

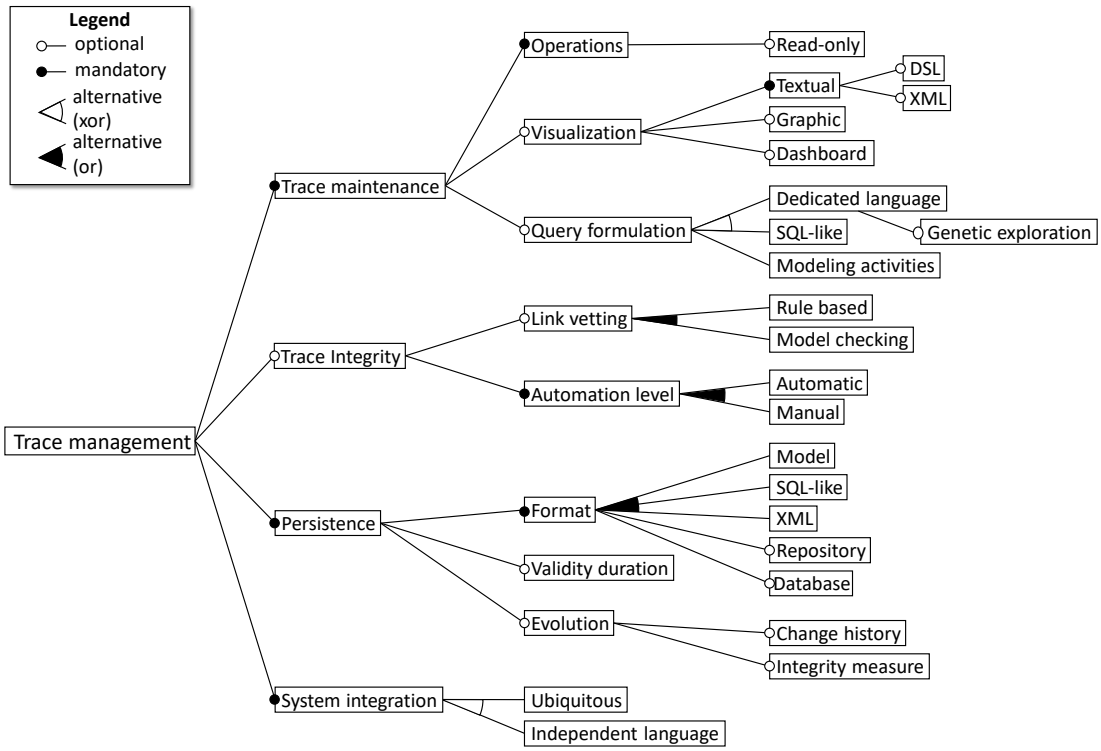

Fig. 5: Tool support for traceability management.

5.4.3 *Trace persistence*

Many different storage alternatives exist for traceability artefacts. An option is to use SQL-like grammar to store and retrieve traces with the power of database tooling, or to use XML documents to represent trace matrix in a transformable format [61]. The industry uses a lot of informal format and link representations often remain implemented in spreadsheets, text files, databases or requirement management tools. These links deteriorate quickly during a project as time pressured team members fail to update them. Researchers aiming at a generalizable approach favour model-based representations able to express specifically defined concepts related to traceability (often in a specific domain of application). The burden of maintaining traces coherent is eased in model-based solutions [21]. Elamin *et al.* propose to implement traceability artefact in graph based databases to improve software quality [29].

Another concern lies in the recording of trace evolution. The trace creation should be recorded, with the successive changes that affect it, for evolution analysis. Integrity measures respective to evolution events (*e.g.,* creation, modification) should be recorded as well to evaluate their evolution during a period of time. Rahimi *et al.* ensure the co-evolution of artefacts and traces [84] using a set of heuristics coupled with refactoring detection and information retrieval technique to detect change scenarios between contiguous versions of software systems.

5.4.4 *System integration*

Like most of the MDE approaches, Helming *et al.* use of the same modeling language for both traceability and system artefacts to track changes [45]. The conjunct use of EMF and a dedicated traceability metamodel (both written in Ecore) facilitates the integration of traceability features including graphical versions to stimulate human perception and standard analysis of traces in the native (Ecore) environment of the traced system.

Galvao *et al.* in their seminal work on traceability and MDE call for more loosely coupled traceability support that can integrate external relationship with independent representations (in another, ideally common language) [33] as also elaborated by Azevedo *et al.* [8].

6 Discussion

The feature model is a first step towards the shared understanding of all dimensions involved in a traceability solution. Ideally, a company interested in a certain set of such dimensions could try to create its perfect traceability solution by combining the top solutions for each dimension. But this is not yet a real possibility as those solution would be difficult to combine and, more importantly, several of the features in the feature model do not really have a great solution yet. This section elaborates on this discussion by presenting some open challenges in software traceability research.

Common traceability metamodel. We have counted over 20 different metamodel proposals. Some are solutions to specific problems the authors present as case studies. And these metamodels are rarely reused, if ever. This proliferation is a challenge to make different traceability solutions interoperate. The research community should agree in a unified proposal that facilitates the composability of traceability solutions. We believe Eclipse Capra [44], even though built to address software product line tracing, could provide a solid foundation for model-based software engineering traceability as it already comes with good tool support to build on. It offers an expressive, yet intuitive customization mechanisms to adapt the tool to any purpose (specific relationship definition) and includes before hand adapters for many Eclipse

elements through EMF ubiquitous integration (artefacts granularity). It also presents traces in different format (graphical, tabular and textual).

Complete traceability metamodel. Following up on the previous point, to agree on a core traceability representation may not seem difficult but it would ignore many of the aspects in the feature model that we believe are key in any non-trivial and industrial traceability application, such as the quality and temporal annotation of traces. A core model with an extension mechanism could be a good compromise here.

Security of trace data. Considering that traceability is a major aspect in certification and other critical applications, it is surprising to see very little interest in security concerns related to trace artefacts. We believe security mechanisms (even simple rule-based access control) for traceability are needed to control who can modify what trace data, given the implication such changes can have.

Library of trace types and semantics. We already mentioned the importance of having a rich set of types for traces to let engineers express the reasons behind the creation of a given trace. But at the same time, complete freedom makes reusability of analysis techniques difficult. We would like to see a rich yet predefined set of types for traces that could then be imported in new traceability projects.

Usefulness of identified traces. Managing a large number of traces is time consuming. As such, we should make sure every explicit trace is actually useful. So far, algorithms aimed at automatically identifying traces are compared based on standard properties like precision and recall. But they should be evaluated on "usefulness": are those traces useful for the end-user? or are just redundant noise?

Verification, validation and testing of traces. Our ample literature on verification, validation and testing methods for software engineering should be extended to deal with trace data, especially from a temporal perspective, where temporality would depend on pure timestamp values (i.e. how long since the trace was created) and on evolution lag (i.e. how many times the linked artefacts have changed since the trace was created). Reasoning on outdated and potentially incorrect trace data could have strong damaging impacts on the system as a whole. So far, very few approaches target these aspects except for the specific problem of coevolution in model-driven engineering. The ability to justify – with evidences and uncertainty evaluation – the quality and integrity of traces is a prerequisite to robust and reliable traceability. And given the effort required to create traces in the first place, this is important to instill more confidence to practitioners wondering whether creating traces is worthwhile.

Traceability as first-class concern in general languages. Another important step towards the mainstream adoption of traceability in industry is the integration of the common traceability metamodel in popular modeling languages like UML or SysML, in the form of a profile (to be able to directly reuse existing modeling tools available for those languages) or new packages in the respective standards. This way, traceability would become a first-class citizen in software development while still being a rich concept and not just the plain dependency relationship we can use right now in those languages.

Working together with Industry. Orthogonal to all the others, we (the research community) should aim to have more frequent exchanges with practitioners to better understand why they end up creating traces manually instead of trying to reuse any of the dozens existing solutions covered in our survey. Some reasons have been already hinted in this paper, based on our own experience in industrial projects involving some type of traceability need and based on the survey we have conducted, but there could be others we are not aware of. Or a different prioritization than the one we have in mind. If we want traceability research to transfer to industry, more and better communication flows should be part of the agenda.

7 Conclusion

Our survey reveals a continuous interest in traceability even if, often, it does not have the spotlight it deserves[1] given the key role it plays in a number of software engineering tasks. Work relating to traceability is indeed disseminated within established research communities (e.g., debugging, SPL). Existing conceptualizations vary greatly depending on the community to which its authors belong to as well as the objectives they aim at. As a consequence, a clear and measurable idea of the costs and benefits to software traceability is slow to emerge

To help visualize, classify and compare the different traceability approaches, we propose a feature model covering all important traceability aspects, as derived from a thorough analysis of the traceability literature. Following the existing body of work, we put special emphasis in separating how traces are represented from how they are identified and managed.

Beyond the feature model, our analysis highlights several limitations of current traceability approaches that should be further developed. Especially given the new challenges the growing use of AI in Software Engineering [94, 103] is introducing (e.g. in terms of reproduciblity and explainability of the AI decisions). In this sense, we hope this paper serves as a "wake-up call" to make sure new proposal comes together with a proper traceability mechanism that assists engineers in recording and understanding the impact of the new AI components in the software engineering process.

As further work, we plan to work on some of the roadmap items above, starting with the proposal of a general traceability metamodel (kind of a superset of all the surveyed ones) that could be used as a starting point in any new traceability project. To facilitate the reuse of such metamodel, we will also release the modeling infrastructure to adapt/refine/deploy it. Once we have this core element, we plan to start working with some of the authors of other proposals to map and bridge their algorithms and techniques to this "unified" metamodel and study how to embed it in other modeling languages (like UML or SysML) to further facilitate its adoption.

References

[1] The DBLP advisory board. The dblp team: Monthly snapshot release of july 2020. DBLP - Computer science bibliography., July 2020. https://dblp.org/xml/release/dblp-2020-0701.xml.gz.

[2] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Systems Journal*, 45(3): 515–526, 2006.

1. As an example, a trace-based paper was awarded the most influential paper in the past 10 years at ICSE [53]. The work introduced a novel trace-based approach to debugging. Yet, the focus was on the debugging aspect of the paper even if traceability was the key to achieve that debugging improvement. The word "trace" alone is mentioned 46 times in the 10 pages paper.

[3] B Amar, H Leblanc, B Coulette, and P Dhaussy. Automatic co-evolution of models using traceability. *Communications in Computer and Information Science*, 170, 2013.

[4] Nicolas Anquetil, Uirá Kulesza, Ralf Mitschke, Ana Moreira, Jean-Claude Royer, Andreas Rummler, and André Sousa. A model-driven traceability framework for software product lines. *Software and Systems Modeling*, 9(4):427–451, 2010.

[5] Giuliano Antoniol, Jane Cleland-Huang, Jane Huffman Hayes, and Michael Vierhauser. Grand challenges of traceability: The next ten years. *CoRR*, abs/1710.03129, 2017.

[6] A. Arunthavanathan, S. Shanmugathasan, S. Ratnavel, V. Thiyagarajah, I. Perera, D. Meedeniya, and D. Balasubramaniam. Support for traceability management of software artefacts using natural language processing. In *2016 Moratuwa Engineering Research Conference (MERCon)*, pages 18–23, April 2016.

[7] Hazeline U. Asuncion, Frédéric François, and Richard N. Taylor. An end-to-end industrial software traceability tool. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 115–124. ACM, 2007.

[8] Bruno Azevedo. and Mario Jino. Modeling traceability in software development: A metamodel and a reference model for traceability. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 322–329. INSTICC, SciTePress, 2019.

[9] Omar Badreddin, Arnon Sturm, and Timothy C. Lethbridge. Requirement traceability: A model-based approach. In *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, pages 87–91, Aug 2014.

[10] Elizabeth Bjarnason, Kari Smolander, Emelie Engström, and Per Runeson. A theory of distances in software engineering. *Inf. Softw. Technol.*, 70(C):204–219, February 2016.

[11] Lossan Bondé, Pierre Boulet, and Jean-Luc Dekeyser. *Traceability and Interoperability at Different Levels of Abstraction in Model-Driven Engineering*, pages 263–276. Springer Netherlands, Dordrecht, 2006.

[12] M. Borg, P. Runeson, and L. Brodén. Evaluation of traceability recovery in context: A taxonomy for information retrieval tools. In *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)*, pages 111–120, May 2012.

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

[14] Mario Borillo, Andrée Borillo, Núria Castell, Dominique Latour, Yannick Toussaint, and M. Felisa Verdejo. Applying linguistic engineering to spatial software engineering: The traceability problem. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, page 593–595, USA, 1992. John Wiley & Sons, Inc.

[15] Elke Bouillon, Patrick Mäder, and Ilka Philippow. A survey on usage scenarios for requirements traceability in practice. In *Requirements Engineering: Foundation for Software Quality*, pages 158–173. Springer Berlin Heidelberg, 2013.

[16] Pierre Bourque and Richard E. Fairley, editors. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos, CA, version 3.0 edition, 2014.

[17] Hugo Brunelière, Erik Burger, Jordi Cabot, and Manuel Wimmer. A feature-based survey of model view approaches. *Softw. Syst. Model.*, 18(3):1931–1952, 2019.

[18] Robert Andrei Buchmann and Dimitris Karagiannis. Modelling mobile app requirements for semantic traceability. *Requirements Eng*, 22(1):41–75, jul 2015.

[19] Hendrik Bünder, Christoph Rieger, and Herbert Kuchen. A domain-specific language for configurable traceability analysis. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development*. SCITEPRESS - Science and Technology Publications, 2017.

[20] Sofia Charalampidou, Apostolos Ampatzoglou, Evangelos Karountzos, and Paris Avgeriou. Empirical studies on software traceability: A mapping study. *Journal of Software: Evolution and Process*, n/a(n/a):e2294, 2020. e2294 JSME-19-0120.R2.

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

[22] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software traceability: Trends and future directions. In *Future of Software Engineering Proceedings*, FOSE 2014, page 55–69, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. Traceability management for impact analysis. In *2008 Frontiers of Software Maintenance*, pages 21–30, 2008.

[24] Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Denys Poshyvanyk. Information retrieval methods for automated traceability recovery. *Software and Systems Traceability*, pages 71–98, 2012.

[25] A. M. Debiasi Duarte, D. Duarte, and M. Thiry. Tracebok: Toward a software requirements traceability body of knowledge. In *2016 IEEE 24th International Requirements Engineering Conference (RE)*, pages 236–245, Sep. 2016.

[26] Timothy Dietrich, Jane Cleland-Huang, and Yonghee Shin. Learning effective query transformations for enhanced requirements trace retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 586–591, Nov 2013.

[27] Nikolaos Drivalos, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. Engineering a dsl for software traceability. In Dragan Gašević, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, pages 151–167, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[28] Nikolaos Drivalos-Matragkas, Dimitrios S. Kolovos, Richard F. Paige, and Kiran J. Fernandes. A state-based approach to traceability maintenance. In *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, page 23–30, New York, NY, USA, 2010. Association for Computing Machinery.

[29] R. Elamin and R. Osman. Implementing traceability repositories as graph databases for software quality improvement.

In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 269–276, 2018.

[30] Stefan Feldmann, Konstantin Kernschmidt, Manuel Wimmer, and Birgit Vogel-Heuser. Managing inter-model inconsistencies in model-based systems engineering: Application in automated production systems engineering. In *Software Engineering 2020*, volume 153, pages 105–134, 2019.

[31] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, Sep. 2013.

[32] J. M. Florez. Automated fine-grained requirements-to-code traceability link recovery. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 222–225, May 2019.

[33] Ismenia Galvao and Arda Goknil. Survey of traceability approaches in model-driven engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 313–313, Oct 2007.

[34] A. Gannous and A. Andrews. Integrating safety certification into model-based testing of safety-critical systems. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 250–260, Oct 2019.

[35] Arda Goknil, Ivan Kurtev, Klaas van den Berg, and Wietze Spijkerman. Change impact analysis for requirements: A metamodeling approach. *Information and Software Technology*, 56(8):950 – 972, 2014.

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

[37] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. *Traceability Fundamentals - Software and Systems Traceability*, pages 3–22. Springer London, London, 2012.

[38] Birgit Grammel, Stefan Kastenholz, and Konrad Voigt. Model matching for trace link generation in model-driven software development. In Robert B. France, Jürgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems - 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30-October 5, 2012. Proceedings*, volume 7590 of *Lecture Notes in Computer Science*, pages 609–625. Springer, 2012.

[39] Victor Guana and Eleni Stroulia. End-to-end model-transformation comprehension through fine-grained traceability information. *Softw Syst Model Systems Modeling*, 18(2):1305–1344, jun 2017.

[40] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 3–14. IEEE Press, 2017.

[41] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2019.

[42] Saida Haidrar, Adil Anwar, and Ounsa Roudies. Towards a generic framework for requirements traceability management for SysML language. In *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*. IEEE, oct 2016.

[43] Abel Hegedus, Gabor Bergmann, Istvan Rath, and Daniel Varro. Back-annotation of simulation traces with change-driven model transformations. In *2010 8th IEEE International Conference on Software Engineering and Formal Methods*. IEEE, sep 2010.

[44] Philipp Heisig, Jan-Philipp Steghöfer, Christopher Brink, and Sabine Sachweh. A generic traceability metamodel for enabling unified end-to-end traceability in software product lines. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 2344–2353, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Jonas Helming, Maximilian Koegel, Helmut Naughton, Joern David, and Aleksandar Shterev. Traceability-based change awareness. In *Model Driven Engineering Languages and Systems*, volume 5795, pages 372–376. Springer Berlin Heidelberg, 10 2009.

[46] Jorg Holtmann, Jan-Philipp Steghöfer, Michael Rath, and David Schmelter. Cutting through the jungle: Disambiguating model-based traceability terminology. In *2020 IEEE 28th International Requirements Engineering Conference (RE)*, pages 8–19, Aug 2020.

[47] Institute of Electrical and Electronics Engineers (IEEE). Ieee guide for software requirements specifications. *IEEE Std 830-1984*, pages 1–26, Feb 1984.

[48] Institute of Electrical and Electronics Engineers (IEEE). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, Dec 1990.

[49] ISO. Road vehicles – Functional safety, 2011.

[50] Álvaro Jiménez, Juan M. Vara, Verónica A. Bollati, and Esperanza Marcos. Model-driven development of model transformations supporting traces generation. In *Building Sustainable Information Systems*, pages 233–245. Springer US, 2013.

[51] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. Form: A feature-;oriented reuse method with domain-;specific reference architectures. *Annals of Software Engineering*, 5(1):143, 1998.

[52] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering – a systematic literature review. *Information and Software Technology*, 51(1):7 – 15, 2009. Special Section - Most Cited Articles in 2002 and Regular Research Papers.

[53] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, page 301–310, New York, NY, USA, 2008. Association for Computing Machinery.

[54] Sahar Kokaly, Rick Salay, Marsha Chechik, Mark Lawford, and Tom Maibaum. Safety case impact assessment in automotive software systems: An improved model-based

approach. In *Lecture Notes in Computer Science*, pages 69–85. Springer International Publishing, 2017.

[55] Simon Frederick Königs, Grischa Beier, Asmus Figge, and Rainer Stark. Traceability in systems engineering – review of industrial practices, state-of-the-art technologies and new research solutions. *Advanced Engineering Informatics*, 26 (4):924 – 940, 2012. EG-ICE 2011 + SI: Modern Concurrent Engineering.

[56] Thibault Béziers la Fosse, Massimo Tisi, and Jean-Marie Mottu. Injecting execution traces into a model-driven framework for program analysis. In *Software Technologies: Applications and Foundations*, pages 3–13. Springer International Publishing, 2018.

[57] W. Li, J. H. Hayes, F. Yang, K. Imai, J. Yannelli, C. Carnes, and M. Doyle. Trace matrix analyzer (tma). In *2013 7th International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 44–50, May 2013.

[58] MIKAEL Lindval and KRISTIAN Sandahl. Practical implications of traceability. *Software: Practice and Experience*, 26(10):1161–1180, 1996.

[59] Patrick Mader, Ilka Philippow, and Matthias Riebisch. A traceability link model for the unified process. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*, volume 3, pages 700–705, July 2007.

[60] Patrick Mader, Orlena Gotel, and Ilka Philippow. Motivation matters in the traceability trenches. In *2009 17th IEEE International Requirements Engineering Conference*, pages 143–148, Aug 2009.

[61] Jonathan I. Maletic, Michael L. Collard, and Bonita Simoes. An xml based approach to support the evolution of model-to-model traceability links. In *Proceedings of the 3rd International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '05, page 67–72. Association for Computing Machinery, 2005.

[62] Ana Cristina Marcén, Raúl Lapeña, Oscar Pastor, and Carlos Cetina. Traceability link recovery between requirements and models using an evolutionary algorithm guided by a learning to rank algorithm: Train control and management case. *J. Syst. Softw.*, 163:110519, 2020.

[63] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 125–135, May 2003.

[64] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *2009 ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, May 2009.

[65] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE Software*, 30(3):50–57, 2013.

[66] N. Mustafa and Y. Labiche. The need for traceability in heterogeneous systems: A systematic literature review. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 305–310, July 2017.

[67] Patrick Mäder, Olive Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *2008 16th IEEE International Requirements Engineering Conference*, pages 23–32, Sep. 2008.

[68] Shiva Nejati, Mehrdad Sabetzadeh, Davide Falessi, Lionel Briand, and Thierry Coq. A sysml-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. *Information and Software Technology*, 54(6):569 – 590, 2012. Special Section: Engineering Complex Software Systems through Multi-Agent Systems and Simulation.

[69] Christian Nentwich, Licia Capra, Wolfgang Emmerich, and Anthony Finkelsteiin. Xlinkit: A consistency checking and smart link generation service. *ACM Trans. Internet Technol.*, 2(2):151–185, May 2002.

[70] C. Neumuller and P. Grunbacher. Automating software traceability in very small companies: A case study and lessons learne. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 145–156, Sep. 2006.

[71] Antoni Olivé. Representation of generic relationship types in conceptual modeling. In Anne Banks Pidduck, M. Tamer Ozsu, John Mylopoulos, and Carson C. Woo, editors, *Advanced Information Systems Engineering*, pages 675–691, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[72] Avital Oliver, Augustus Odena, Colin Raffel, Ekin D. Cubuk, and Ian J. Goodfellow. Realistic evaluation of deep semi-supervised learning algorithms. *CoRR*, abs/1804.09170, 2018.

[73] Richard Paige, Gøran Olsen, Dimitrios Kolovos, Steffen Zschaler, and Christopher Power. Building model-driven engineering traceability classifications. In *Computer Science*, 01 2010.

[74] Richard F. Paige, Nikolaos Drivalos, Dimitrios S. Kolovos, Kiran J. Fernandes, Christopher Power, Goran K. Olsen, and Steffen Zschaler. Rigorous identification and encoding of trace-links in model-driven engineering. *Software & Systems Modeling*, 10(4):469–487, 2011.

[75] Richard F. Paige, Athanasios Zolotas, and Dimitris Kolovos. The changing face of model-driven engineering. In *Present and Ulterior Software Engineering*, pages 103–118. Springer International Publishing, 2017.

[76] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 522–531, May 2013.

[77] Annibale Panichella, Collin McMillan, Evan Moritz, Davide Palmieri, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. When and how using structural information to improve ir-based traceability recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 199–208, March 2013.

[78] Michael C. Panis. Successful deployment of requirements traceability in a commercial engineering organization...really. In *2010 18th IEEE International Requirements Engineering Conference*, pages 303–307, Sep. 2010.

[79] A. Paz and G. El Boussaidi. A requirements modelling language to facilitate avionics software verification and certification. In *2019 IEEE/ACM 6th International Workshop on Requirements Engineering and Testing (RET)*, pages 1–

8, May 2019.

[80] Francisca Pérez, Tewfik Ziadi, and Carlos Cetina. Utilizing Automatic Query Reformulations as Genetic Operations to Improve Feature Location in Software Models. *IEEE Transactions on Software Engineering*, 2020.

[81] Rolf-Helge Pfeiffer, Jan Reimann, and Andrzej Wąsowski. Language-independent traceability with lassig. In *Modelling Foundations and Applications*, pages 148–163. Springer International Publishing, 2014.

[82] S.L. Pfleeger and S.A. Bohner. A framework for software maintenance metrics. In *Proceedings. Conference on Software Maintenance 1990*, pages 320–327, Nov 1990.

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

[84] M. Rahimi and J. Cleland-Huang. Evolving software trace links between requirements and source code. In *2019 IEEE/ACM 10th International Symposium on Software and Systems Traceability (SST)*, pages 12–12, May 2019.

[85] Brian Randel. Towards a methodology of computing system design. *NATO Software Engineering Conference*, Brussels, Scientific Affairs Division, NATO (Published 1969):pp. 204–208, 1968.

[86] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. Traceability in the wild: Automatically augmenting incomplete trace links. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 834–845, New York, NY, USA, 2018. Association for Computing Machinery.

[87] Hajer Saada, Marianne Huchard, Clementine Nebut, and Houari Sahraoui. Recovering model transformation traces using multi-objective optimization. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, nov 2013.

[88] Iván Santiago, Juan M. Vara, Valeria de Castro, and Esperanza Marcos. Measuring the effect of enabling traces generation in ATL model transformations. In *Communications in Computer and Information Science*, pages 229–240. Springer Berlin Heidelberg, 2013.

[89] Iván Santiago, Juan Manuel Vara, María Valeria de Castro, and Esperanza Marcos. Towards the effective use of traceability in model-driven engineering projects. In Wilfred Ng, Veda C. Storey, and Juan C. Trujillo, editors, *Conceptual Modeling*, pages 429–437, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[90] Iván Santiago, Álvaro Jiménez, Juan Manuel Vara, Valeria De Castro, Verónica A. Bollati, and Esperanza Marcos. Model-driven engineering as a new landscape for traceability management: A systematic literature review. *Information and Software Technology*, 54(12):1340 – 1356, 2012. Special Section on Software Reliability and Security.

[91] Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Software & Systems Modeling*, 9(4):473–492, 2010.

[92] Andreas Seibel, Stefan Neumann, and Holger Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software & Systems Modeling*, 9(4):493–528, 2010.

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

[94] Saad Shafiq, Atif Mashkoor, Christoph Mayr-Dorn, and Alexander Egyed. Machine learning for software engineering: A systematic mapping, 2020.

[95] Y. Shin, J. H. Hayes, and J. Cleland-Huang. Guidelines for benchmarking automated software traceability techniques. In *2015 IEEE/ACM 8th International Symposium on Software and Systems Traceability*, pages 61–67, May 2015.

[96] Oscar Slotosch and Mohammad Abu-Alqumsan. Modeling and safety-certification of model-based development processes. In Ina Schaefer, Dimitris Karagiannis, Andreas Vogelsang, Daniel Méndez, and Christoph Seidl, editors, *Modellierung 2018*, pages 261–273, Bonn, 2018. Gesellschaft für Informatik e.V.

[97] George Spanoudakis, Andrea Zisman, Elena Pérez-Miñana, and Paul Krause. Rule-based generation of requirements traceability relations. *Journal of Systems and Software*, 72 (2):105 – 127, 2004.

[98] Claudia Szabo and Yufei Chen. A model-driven approach for ensuring change traceability and multi-model consistency. In *2013 22nd Australian Software Engineering Conference*. IEEE, jun 2013.

[99] Bedir Tekinerdoğan, Christian Hofmann, Mehmet Akşit, and Jethro Bakker. Metamodel for tracing concerns across the life cycle. In Ana Moreira and John Grundy, editors, *Early Aspects: Current Challenges and Future Directions*, pages 175–194, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[100] Tassio Vale, Eduardo Santana de Almeida, Vander Alves, Uirá Kulesza, Nan Niu, and Ricardo de Lima. Software product lines traceability: A systematic mapping study. *Information and Software Technology*, 84:1 – 18, 2017.

[101] Juan Manuel Vara, Verónica Andrea Bollati, Álvaro Jiménez, and Esperanza Marcos. Dealing with traceability in the mddof model transformations. *IEEE Trans. Software Eng.*, 40(6):555–583, 2014.

[102] A. von Knethen. Change-oriented requirements traceability. support for evolution of embedded systems. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 482–485, Oct 2002.

[103] Cody Watson, Nathan Cooper, David Nader Palacio, Kevin Moran, and Denys Poshyvanyk. A systematic literature review on the use of deep learning in software engineering research, 2020.

[104] Duncan J. Watts. Should social science be more solution-oriented? *Nature Human Behaviour*, 1(1):0015, 2017.

[105] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9(4):529–565, 2010.

[106] Rebekka Wohlrab, Eric Knauss, Jan-Philipp Steghöfer, Salome Maro, Anthony Anjorin, and Patrizio Pelliccione. Collaborative traceability management: a multiple case study from the perspectives of organization, process, and culture. *Requirements Engineering*, 25(1):21–45, 2020.

[107] Dennis Ziegenhagen., Andreas Speck., and Elke Pulvermüller. Using developer-tool-interactions to expand tracing capabilities. In *Proceedings of the 14th International Con-*

ference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,, pages 518–525. INSTICC, SciTePress, 2019.

[108] Dennis Ziegenhagen, Andreas Speck, and Elke Pulvermueller. Expanding tracing capabilities using dynamic tracing data. In *Communications in Computer and Information Science*, pages 319–340. Springer International Publishing, 2020.