Taming Cross-Tool Traceability in the Wild

Cosmina Cristina Rațiu*, Christoph Mayr-Dorn*, Wesley K. G. Assunção*, Alexander Egyed*
*Institute of Software Systems Engineering
Johannes Kepler University Linz
Linz, Austria

***Abstract*—Along the process of engineering a safety-critical system, software engineers produce various artifacts, ranging from requirements and change requests to source code and test cases. In order to aid the development of the system and to adhere to the complex safety regulations and standards in place, engineers are often required to maintain bidirectional and consistent traceability between the produced artifacts. However, such artifacts are rarely maintained in one single tool. Because of that, the cross-tool bidirectional traces have to frequently be manually maintained, which can easily become a very time-consuming or infeasible task. Through interviews and observations at our industry partners in regulated domains, we observed that a number of different strategies are used to deal with this challenge. The use of naming conventions, querying, or URL links is observed in the industry. However, they have their shortcomings and hinder engineers from realizing the full potential that traceability can offer. Knowing the challenges in the industry, we explored existing literature. A range of approaches in the literature aims at dealing with traceability, but often they are context-specific and not easily transferable into practice. Given this gap between the state-of-the-art and industry needs, we performed interviews with our industry partners and analyzed tertiary studies from the literature to obtain a better understanding of what traceability properties are needed to unleash the potential of traces. We identified properties that represent the shared challenges between the related work and the industry requirements: discoverability, type checks, flexibility, navigability, and extensibility. While each property is addressed by a subset of the available solutions, we propose a novel traceability approach to support all of them in a single tool.**

***Index Terms*—Traceability, Regulation, Industry needs, Artifact linking**

I. Introduction

In safety-critical systems or regulation-centric domains (e.g., aerospace, automotive, and robotics), the software development process needs to produce traces between various artifact. The need for such traces is to validate/demonstrate that the system under development has expected quality [1]–[3]. Some examples of the standards that require extensive traceability throughout the process, from user requirements down to test results, are the ASPICE standard applied in the automotive domain [4]–[7], DO-178C/ED-12C for airborne systems [8], or ED-109A for air traffic management systems.

Due to the complexity of developing such safety-critical and regulated systems [9], [10], often variant-rich systems to meet the requirements of many end users [11], [12], companies typically utilize a diverse set of tools for managing the artifacts created across the different stages of the development process [13]. However, there is no *de facto* standard to keep track of related artifacts created with different tools. Each tool offers different means to support traceability [14]. For example, some tools allow explicit traces to artifacts beyond their tool boundary via an OSLC API, other cases require the usage of third-party plugins that manages the traces and links into the tools on either side of a trace link. Other traces are manually managed by copying and pasting identifiers or URLs into dedicated fields in the tools on both ends of a trace link. Furthermore, discussions with our industry partners have revealed that often a combination of these approaches is necessary, which in turn increases the complexity of the traceability process. Thus, maintaining consistent traceability became a cumbersome, complex, and time-consuming process.

To make matters worse, establishing traces is perceived as an unnecessary overhead by developers and rarely as an activity they will benefit from [15]. The only perceived reason for having consistent and complete traces is regulatory requirements. Surprisingly, realizing additional benefits such as impact assessment or test prioritization is unexplored in the automotive industry [16]. Hence, reducing the effort to create and maintain traces while making them more amenable to automated processing and navigation is a key aspect towards increasing the efficiency and impact of traceability.

As part of the study reported in this paper, we conducted interviews with engineers at one of our industry partners to understand the practical implications and challenges of traceability. Next, we compared their traceability approach with our observations at other industry partners, as well as those described in the literature [16]. Then, the observed challenges, including those found in the literature, motivated and guided the design of our novel approach for dynamic trace specification and navigation, including a prototype implementation used as a proof of concept.

The primary innovation of our approach is the dynamic integration of trace information into the artifact representation for seamless navigability. In other words, rather than having traces only as separate artifacts, we dynamically augment the artifact representation (i.e., the artifact's metamodel) with a trace specification, thereby making trace links available directly from the artifact instances themselves. One advantage of automated processing of traces is no longer requiring manual navigation across explicit traces or resolving of URLs to obtain the counterpart artifacts on which to continue navigation.

II. Motivation and Problem Statement

In this section, we describe traceability challenges and strategies found in the industry while dealing with different

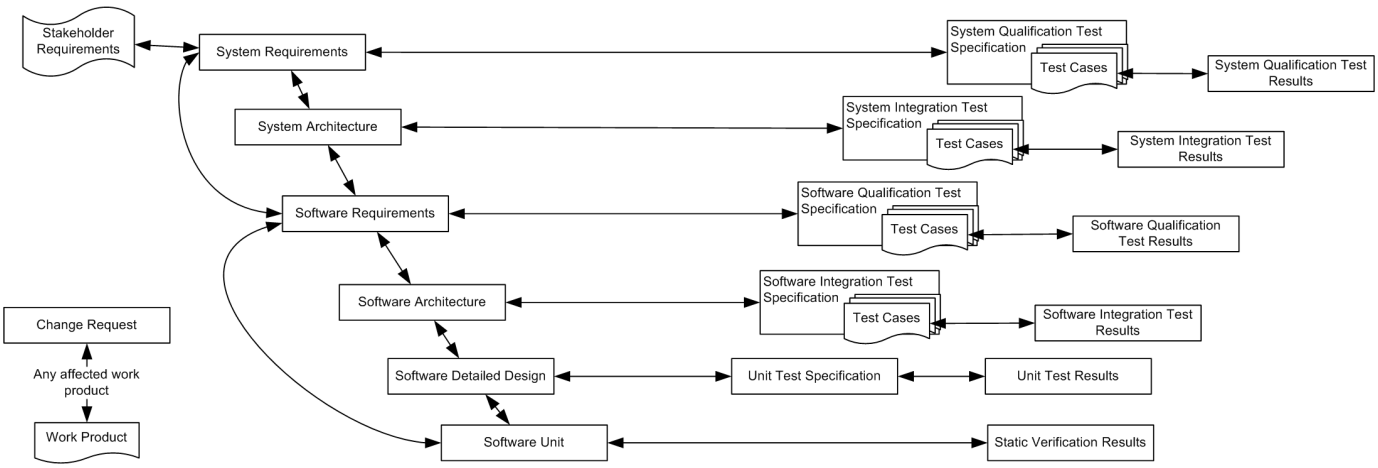

Fig. 1. Traceability Information Model (TIM) mandated by ASPICE (recreated from ASPICE v3.1 Fig D.4).

artifacts and tools. As previously mentioned, our observations are based on seven interviews[1] with our industry partners, during which engineers involved in the software development process expressed their experiences and challenges when dealing with traceability.[2] Also, we highlight the limitations of existing traceability support in practice. Based on that, we describe the desired properties that support compliance with existing traceability regulations (e.g., ASPICE) while providing additional benefits for the engineering tasks such as impact assessment and test prioritization.

For the sake of completeness, we describe a generic engineering process in a change-driven development environment. The development process for our industry partners, and other safety-critical system development, typically starts with a *change request* coming from a client. For example, the client may require an increase in the range of a particular configuration parameter. Assuming the change request is approved, the engineering team will determine the *software requirements* that are affected, those that also need updating, and any additional ones that need to be created. These requirements then also need to be traced to higher level *system requirements* within the requirements' management tool and also to the change request in the issue tracking tool. When both System and Software Requirements are managed in the same tool, establishing and managing bidirectional, navigable traces between them is not a significant challenge. However, tracing across tools, e.g., from the requirements (e.g., in IBM Rational DOORS) to the change request (e.g., in Jira), is not as straightforward without additional support. A common solution in the industry involves the use of two URL links, on each side, to point to the respective artifact in the other tool.

Based on the updated software requirements, the development team changes and extends the *models*. The overall set of models makes up the system design. Design and Implementation activities occur in a wide variety of tools. During design, engineers create a specifications document for each implementation aspect, which can use textual flow documentation, high-level model fragments, and links to the requirements covered by different parts of the implementation.

Discoverability. To comply with the traceability regulations, models (or architectural elements) must trace to the requirements they realize. Hence, any changed model may link to new, updated, and existing software requirements. To this end, engineers need to identify which are the respective requirements. When making a change to a large system, there are potentially thousands of requirements and thousands of model elements, which makes the traceability task cumbersome.

Identifying which of the updated and newly created requirements need to trace to a particular model is usually not a great challenge, as engineers typically have these requirements in mind when developing the model. Tracing to other existing, relevant requirements, however, is a more difficult task. As mentioned, due to the size of the system and amount of requirements, it is unlikely that engineers know all the requirements well enough to identify any further trace relations without effort or additional help. This situation highlights the first essential traceability property: *Discoverability*.

> **Discoverability** is the ability of an engineer to become aware of which traces already exist as well as to identify all the possible endpoint artifacts at the desired level of granularity.

Type checks. In addition to the discovery artifacts to trace, traceability regulations mandate traces between specific artifacts (i.e., endpoints), e.g., traces between Software Requirements and Software Units (see Figure 1). To this end, engineers must ensure that they create traces between the right artifact types and avoid traces between artifacts that need not be traced (by accident or an unclear trace process). Thus, discoverability is tightly related to another traceability property: *Type checks*.

[1]We invited participants for the interviews using convenience sampling, i.e., practitioners easily accessible in our industry partners [17].

[2]Note that, due to confidentiality concerns, we cannot describe tools, processes and exact trace types in detail.

Type checks give engineers the ability to ensure that the created traces indeed are those as required by a given regulatory standard.

Engineers in the industry face a challenge for type checks, as the set of tools in use provides type checks that are usually limited. In fact, tools typically allow establishing a trace from any of its managed artifacts to any other artifact type for the simple reason that tools need to be flexible for use in many different development contexts. No type checks are available when simply URLs of artifacts or their identifiers are used to establish the trace (see also the Navigability property below). The lack of type checks may lead to incomplete/incorrect trace information that is then only later found during reviews. One mitigation strategy available for customizable tools, we found with our industry partners, is the deployment of plugins or extensions that restrict or guide engineers during trace creation. However, the development, configuration, and maintenance of these in-house plugins require significant effort and often need to be done for any type of cross-tool trace (e.g., requirements to model and requirements to test cases).

We have an example of the effect of lacking type checks. Most often, there exists a system specification document that describes each system component, including trace links to the related requirements, e.g., by specifying a requirement ID. When this ID is manually added, there are no automatic type checks to ensure it is actually an existing requirement. Thus, inconsistencies are only detected during a manual review process, which is costly and delays the development process.

Type checks support *Discoverability* as they trivially allow filtering based on trace types, filtering the endpoint artifacts to be displayed when creating a trace. Also, type safety might restrict the engineers by limiting the available traces to a subset inadequate for the underlying development process.

Flexibility. Regulations need to be applicable to many different companies and development scenarios. Hence, they describe the traceability information model only at an abstract level (e.g., system requirements or software units) without identifying concrete artifact types and their tools (e.g., a requirement in DOORS or a C++ header file versioned in Git). It is then up to the companies to establish a grounding of these abstract trace definitions to concrete traces and trace types within their tool environment. Thus, the required trace information is ultimately grounded in tools that are typically more generic, rather than specifically made for just that regulatory trace standard. Additionally, large companies and/or complex systems usually come with the need to integrate different tools and artifacts. Different programming languages, different modelling languages, different requirements or change requests, and even different processes require each a separate grounding of the abstract trace definitions. To meet industrial needs, we have an essential traceability property: *Flexibility*.

Flexibility describes the ability to define the set of specific trace types (independent of the utilized tools) at the level of concrete artifacts.

A common example encountered by our industry partners is that different test case to software requirement tracing possibilities exist depending on the representation of the test cases. One option could be in the context of docs-as-code to specify the identification of the requirement by ID in a so-called *ReStructured* file[3] that describes the test case, while the alternative option specifies the use of a Test Case item in Jira.

Similarly, some teams within one organization may generate all their code from models, while other teams may require additional manual coding activities. The traceability between requirements and implementation still has to be ensured, but needs to be done differently. The former kind of team uses the documentation of their models to establish the traces to requirements. The latter kind of team might require a different set of traceability support tools that enables tracing from more fine-grained artifacts, such as classes or even methods.

Ultimately, *Flexibility* enables the benefits of *Type Checks* to be available in many different development contexts within the same organization and toolset. Together with *Discoverability*, these are essential in creating and maintaining traces but not sufficient for efficient consumption.

Navigability. Often it is required to be able to traverse a trace link in both directions (i.e., bidirectional tracing) as, for example, mandated by ASPICE (see Figure 1) and DO-178C.

Continuing with the test case to requirements traceability example above, when test cases are specified in *ReStructured* text files, no explicit traceability support is available. One possibility to establish traceability is by including a URL to the requirement in a specific part of the document. In the opposite direction, one may not want to manage a plethora of URLs to downstream artifacts but rather apply a search function, i.e., RST files are filtered based on the appearance of a requirement URL, a far from ideal solution.

Establishing traces bottom-up as well as right (verification) to left (specification) is a common practice as most processes foresee creating coarse-grained artifacts such as system requirements first to be followed by refinement into finer-level artifacts such as software requirements. However, traceability audits, for example, are done top-down to check that system requirements are covered by system qualification test cases, and are covered by software requirements. Basically, in order to use traces, *Navigability* is an important property.

Navigability describes the ability of an engineer (or also a supporting tool) to traverse the trace graph from one artifact endpoint to another.

Our industry partners have reported that creating navigable bidirectional links is a challenge that requires time and effort to do completely and consistently. The tools they use are not well integrated, such as the traces between requirements and test cases, which often results in engineers creating two unidirectional links at each endpoint that have to be maintained separately. While the requirements can be linked to the test cases through search functions, each test case is provided with

[3]https://en.wikipedia.org/wiki/ReStructuredText

a field which links to the covered requirements through URL links. These two traces then have to be maintained separately.

Beyond human navigability, machine-readable navigability is an even greater challenge. There are potential benefits of trace links beyond regulatory requirements, such as support for change impact analysis [18]–[20], consistency checking [19], [21], change propagation [22], [23], or test case prioritization [24], [25]. For that, tools need to access engineering artifacts, regardless of how these are managed across different tools. While using URL links is sufficient for human navigability, where the engineers can traverse from artifact to artifact by following the links, they pose a challenge to programmatic navigability as a URL typically lacks any hints on what to expect (at the semantic level, beyond a web resource) and how to access that artifact at the URL's endpoint.[4] A supporting mechanism to navigate programmatically across tools has to become aware of not just how to access the various artifact, but also understand their data model.

This challenge remains largely unresolved in practice. Hence, traces are primarily set for documentation purposes only, and many checks that could be otherwise automated are done by manually navigating the links. This implies that engineers are the producers of trace links without benefiting much from their existence. While they understand the importance of traces from a regulatory perspective, they do not see the use of such links from their own perspective. Therefore, their motivation lies in setting traces to fulfil the regulatory requirements rather than traces that can help them automate further process steps. Overall, *Navigability* is challenging by itself in a fixed tool landscape with a stable set of trace types.

Extensibility. Systems in the automotive domain are typically maintained and evolved over longer periods of time. As expected, engineers of such systems experience changes to the regulatory standards, engineering processes, and available tools. These changes often come with the necessity to adapt the mandated trace types, hence the need for *Extensibility*.

> **Extensibility** is the ability to introduce new trace types and trace to novel tools once a project or system has already undergone significant development (and hence tracing) effort.

The introduction of new trace types must not invalidate any preexisting trace instances already available, but be able to maintain new and former trace types at the same time. New engineering tools come with their own metamodel and model elements, which need to be integrated seamlessly within the existing trace model. Not only should these elements be linkable within the model (see *Navigability*), but an extensible trace mechanism should also make the engineer aware of whether new model elements were considered for traceability or not (see *Discoverability*).

One example of adding new trace types by one of our industry partners was the decision to explicitly trace artifacts emerging from functional safety engineering (ISO 26262) efforts. Functional safety-related artifacts include failure mode and effect analysis (FMEA) results or fault tree analysis (FTA) results but could even include detailed artifacts such as individual safety hazards [26]. A safety case then identifies all safety-relevant artifacts via traces. These new trace types need to be supported in any engineering context (also in any tool) that strives to adhere to Functional Safety regulations.

In addition to eliciting the desirable traceability properties with our industry partners, we investigated the state of the art as well as existing open-source or commercial tooling support to understand to what extent companies are facing the same challenges, respectively to what extent these are solved. To this end, we discuss the representative state of the art and tools regarding traceability in the next section.

III. State of the Art and the Practice

This section describes the state of the art and the practice in the topic of our work. At the center is an insightful tertiary study by Maro et al. [16] that reports challenges and solutions of traceability in the automotive domain. For the state of the practice, we describe a popular set of industrial and open-source tools and how they support the properties described in the previous section. The tools were selected based on recommendations and experience reviews coming from our industry partners.

A. Traceability Support Challenges

After collecting the properties desired by our industry partners, we searched for related work in the literature to identify a set of challenges that overlap with our findings. In this section, we describe such an overlap and compare them to the challenges reported by Maro et al. [16], of which an overview is presented in Figure 2.

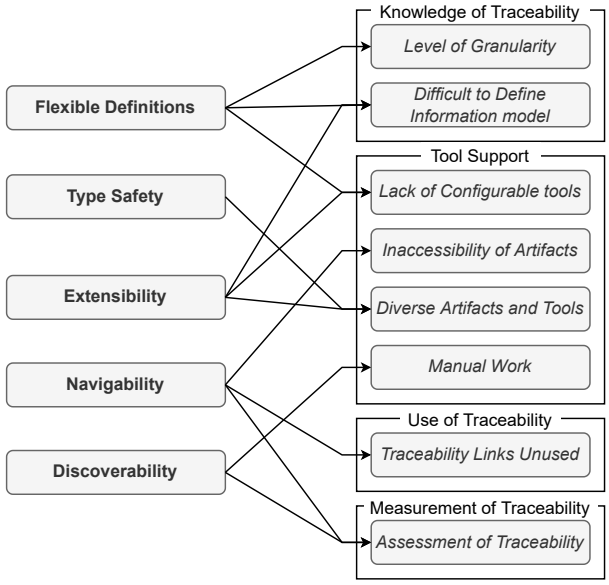

Fig. 2. Required properties observed in our industry partners compared with challenges presented by Maro et al. [16]

[4]OSLC aims to overcome this shortcoming to some degree, but a navigation mechanism still needs to remain aware of the tool boundaries.

We find parallels between the need for **discoverability** and the required *manual work* and *assessment of traceability* challenges posed by Maro et al. [16]. A significant hurdle in the way of addressing these challenges is the effort required to find the artifacts that need to be traced and identify missing traces [27]. While the literature is extensively exploring the automatic generation of trace links and using artificial intelligence in this sense, Fucci et al. [28] highlight that many organizations cannot benefit from these techniques due to the structure and diversity of the artifacts they use. This leaves the problem yet unsolved, being supported by trace visualizations such as matrix views [29]–[31], or even expecting the engineers to look up artifact IDs in other tools, as addressed by Wohlrab et al. [32].

Type checks become a topic of interest as soon as the *diversity of the artifacts and tools* increases to the point where integration is necessary. Tool integration is a major challenge still open in the literature [33], as well as in the tools available in practice [28], [30], [32]. In the industry, this problem is addressed in different ways, from using URL links as presented in Section II, to copying artifact IDs [32]. While these solutions are usable, they allow a level of flexibility that impedes actionable and effective guidance [33].

Further, the variety in artifact types and tools used, even within one company, can lead to *difficulties in defining a traceability information model* (TIM) [16], [34], which is related to **flexible definitions**. The traceability requirements may differ depending on the stakeholder and their specific needs in the project [31], [34], the *granularity level* of the artifacts to be linked [16], [28] and the business processes used by the organization [35], [36]. The result is the need to support increasingly complex traceability information models. Therefore, the tools should be *configurable* enough to allow flexible trace type definitions [33].

The level of complexity does not remain stagnant during the development process, but instead continues to rise as the project and its requirements evolve [34]. Thus, the TIMs are also evolving [16], which may lead to new trace types being added to a TIM in the middle of the development process. Moreover, as the needs of the stakeholders evolve, *new tools with different artifact types* could be introduced, requiring traces to new types of elements [33], [34]. The traceability support tools should not only be configurable enough to allow flexible definitions, but additionally, they should be **extensible** to address changes and additions to these definitions without affecting existent trace links.

Finally, even solving the above issues, a challenge that persists involves the engineers directly, in that they cannot see the benefit in creating and maintaining trace links, as noted by Wohlrab et al. [32]. Indeed, Maro et al. [16] discuss their observation that after creation, most traceability *links remain unused*. Additionally, they mention that, although linked, the artifacts remain *inaccessible* because the traces cannot be properly navigated (**navigability**). The addition of navigable traces not only motivates the need for thorough traceability, but also opens the possibilities to multiple processes being automated, such as *link assessment* [16], [32], coverage analysis and documentation support [30], as well as consistency checks and change impact analysis [27], [33], [35].

B. Traceability Support in the State-of-the-Art Tools

In this subsection, we compare four common traceability tools that we identified via literature research and through informal discussions with our various industry collaborators, including our industry partners. We used the tools' existing documentation to evaluate the extent to which they cover the five traceability support properties we have identified. The results of this analysis can be seen in Table I.

TABLE I
TOOLS SUPPORT FOR TRACEABILITY PROPERTIES IN PRACTICE

Tool	Discoverability	Type Safety	Flexibility	Navigability	Extensibility
CAPRA	●	●	●	○	○
DOORS	◓	◓	○	◓	◓
Polarion	●	●	◓	◓	◓
Reqtify	◓	○	○	●	●
Our approach	●	●	●	●	●

○ Not supported, ◓ Partially supported, ● fully supported

A well-known open-source traceability tool that emerged from the scientific community is Eclipse CAPRA,[5] which is the state-of-the-art traceability support tool in the scientific literature. The main goal in its design is flexibility, and its EMF base is highly supportive of this property. Any artifact type that can be represented in EMF can also be linked in CAPRA. Moreover, the trace links are also typed, assuring additional type checks. However, it must be noted that, despite being extensible, EMF has the disadvantage of requiring regeneration of the underlying datamodel to make use of new customizations. Therefore, CAPRA's extensibility cannot be achieved at runtime. Hence, a lot of effort is required to create a suitable trace type configuration before traceability starts. This configuration in itself is highly flexible, but new required trace types would be more cumbersome to integrate. Additionally, CAPRA offers a range of visualizations, but most of them are geared towards analyzing the existent links, and less toward aiding the user in discovering possible endpoints for new traces. Finally, the traceability model is stored as an EMF model, but by default, CAPRA does not offer any programmatic navigability features over the created links.

Next, we have considered IBM Rational DOORS,[6] a tool our industry partners are already using in part in their requirement engineering processes. This tool supports three different kinds of links: internal, external, and collaboration links. The internal links can only connect the artifacts created in the same DOORS database. As our solution focuses on cross-tool traceability, these links fall outside our scope. We are therefore only considering external and collaboration links in our discussion. The external links connect the DOORS artifacts with any

[5]https://projects.eclipse.org/projects/modeling.capra
[6]https://www.ibm.com/products/requirements-management

artifact outside the DOORS database, through URL links. This solution partially covers the problems of navigability. These links can both be easily used by practitioners for navigation and can be parsed to result in a level of programmatic navigability that could fulfil part of the use cases for traceability. However, type checks cannot be assured as any element can be linked as long as a URL link can be obtained. This also leads to discoverability being impossible to assure. On the other hand, collaboration links trace the DOORS requirements to artifacts created in tools which support OSLC. This results in links that are typed, allowing for type checks, as well as discoverable, by allowing the visualization of the linkable elements in the target tool. DOORS also allows the navigation of these OSLC links manually through previews when hovering over a link, but not programmatically without additional services added to the default configuration. However, each tool that is integrated with OSLC only provides a specific set of link types, which cannot be expanded easily with user-created traceability types. This results in lower levels of extensibility and flexibility in the definitions of the links.

A similar tool that is often used in industry is Siemens Polarion,[7] which also focuses on requirement management, but allows tracing the requirements to different artifacts within a system. The traces are typed, with a variety of trace types being available for different combinations of source and target types. This assures type checks to a high degree, but also reduces the flexibility for defining new trace types. From the perspective of the endpoints, a listener reacts to any new instances of an endpoint type and they are immediately available for tracing. Additionally, multiple visualization options allow the discovery of traceable instances and a view of the trace status in the system. Yet, Polarion can use the created traces to perform change impact analysis and coverage analysis, implying that the links are programmatically navigable.

Finally, Reqtify[8] is another requirement management tool frequently used in the industry, in terms of traceability to requirements. The traces created in this tool document the requirement coverage by test cases and other relevant artifacts. As such, the endpoints of these links are not type-checked, resulting also in less flexible definitions, in that no other trace types besides coverage can be specified. However, new elements are automatically added to the tree view of traceable endpoints, resulting in a highly extensible solution, which also allows for easy discovery of possible traced elements. Additionally, Reqtify allows a full programmatic navigation of the trace network to determine the requirement coverage status.

While this set of tools is just an excerpt, to the best of our knowledge they represent the tool capabilities available for software and systems engineering. As highlighted in Table I none of them fully covers all the properties desired by commercial development in general, and our industry partners in particular. To pave the way in the direction of having such a kind of tool support, this paper describes our proposed approach, described in the following sections.

[7]https://polarion.plm.automation.siemens.com/

[8]https://www.3ds.com/products-services/catia/products/reqtify/

IV. Approach

Our traceability approach addresses the five traceability properties through a combination of mechanisms (see Figure 3). We briefly introduce these mechanisms below and discuss them in more detail in the following subsections.

A. Architecture Overview

The primary design decision is a centralized collection of engineering artifacts with a flexible data model. Each *Tool Connector* (A) describes the artifacts in the respective tool independently of any other tool. The *Trace Type Specification* refers to these data models to specify which artifact type traces to which other artifact type (B), thereby resulting in a dynamically generated *Trace Matrix* for each trace type (C). Changes in tools to an artifact (i.e., creation, deletion, and update) are pushed into the *Artifact Store* (D). This includes changes to tool-internally defined traces that need to be translated into our internal trace model. These changes result in events processed by the *Trace Matrix Maintainer* (E) that ensure that each affected trace matrix contains an up-to-date list of artifacts by shrinking/expanding the matrix (F). These change events are also used by *Link Translators* to import manually managed cross-tool trace (G) and *Trace Analysis Services and Plugins* (J) to check for likely missing or incorrect traces. Creating new and correcting existing traces occurs via the *Trace Matrix Dashboard* (H). When possible, these traces are propagated back to the respective tools via the *Link Translators* (I).

B. Engineering Artifact Integration

The first step in providing traceability support is finding a common space where all the artifacts are brought together. In our approach, we achieve this by using *DesignSpace*[9] as the underlying artifact store [37]. *DesignSpace* stores all artifacts uniformly, allowing them to be further linked and reasoned with beyond the boundary of their origin tool, while also keeping them synchronized with their in-tool representation.

In order to achieve this uniformity, *DesignSpace* stores each model together with its respective metamodel. The metamodel is represented as artifact type elements, called InstanceTypes (see Figure 4 top). *InstanceTypes* specify the structure of the respective artifacts (i.e., the available artifact properties), as well as the relations they have to other elements within the tool they are created in. In Figure 4 we depicted a *RequirementType* and *C++Type* (without detailing any properties or relations for sake of clarity) that represent a requirement from a tool like DOORS, and a C++ class file, respectively, available via GitHub, for example. Then, each specific artifact is an *Instance* of one *InstanceType*, with its structure adhering to that type and hence its defining information accessible via the defined *Properties*. This uniformity balances *flexible definitions* with

[9]https://isse.jku.at/designspace/

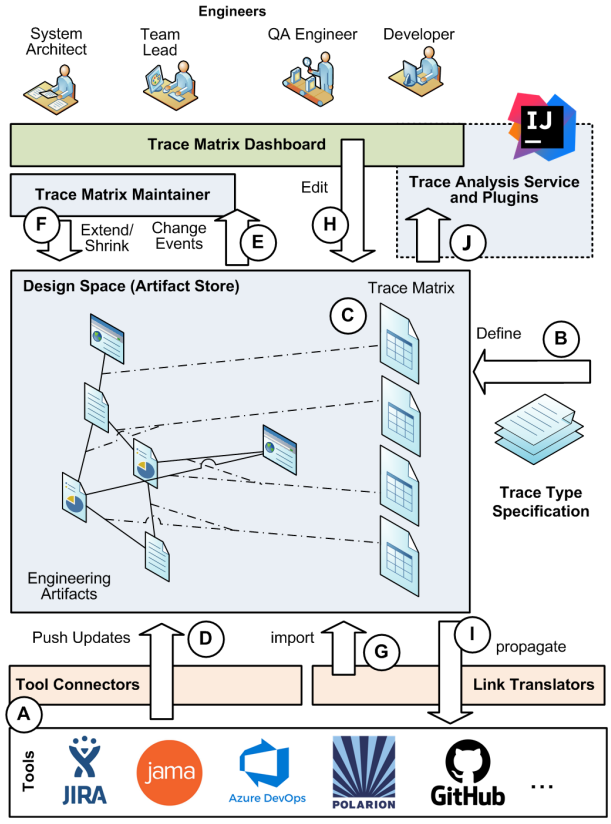

Fig. 3. Approach Overview

type safety. Figure 4 bottom depicts a few requirements and C++ class instances.

In order to make the artifacts in the various tools available in *DesignSpace*, a *Tool Connector* is required for each tool, similar to how EMF-supporting tools are integrated into Eclipse CAPRA. These adapters determine which elements are available within a tool and translate between the tool's artifact representation (i.e., the tool's metamodel) and the artifact representation (i.e., *InstanceType*) in *DesignSpace*. Any change to an *Instance*, whether it is happening in the tool or in the artifact store, is synchronized via the *Tool Connector*. However, an advantage compared to EMF that contributes towards *extensibility* is the fact that new *Tool Connectors* and hence also new *InstanceType* definitions can be added at runtime without requiring additional integration steps. This includes the ability to change existing type definitions.

C. Traceability Data Model

The trace type specification metamodel within *DesignSpace* uses the same structure as the artifact metamodels. Thus, trace types are a distinct set of *InstanceTypes*, which inherit from a generic *TraceMatrixType* element (see Figure 4 top). Each trace type is defined by its name and the types of the two endpoints it connects (i.e., two *InstanceTypes*). The two *InstanceTypes* are referred to via the *RowType* and *ColumnType* that make up the matrix. Therefore, *DesignSpace* assures traces are *type checked* in that the approach does not allow a trace to link to any other type than the one defined in its trace type definition. The two endpoint types can be any *InstanceType* available in *DesignSpace*, resulting in complete *definition flexibility*. In Figure 4 we depict a requirements-to-code matrix type (*ReqToCodeMatrixType*) pointing to the earlier defined *RequirmentType* and *C++Type*. The TraceMatrixType further defines that each traceability matrix instance has a set of *Rows* and *Columns* that are kept in sync to represent the matrix structure. A row does not simply contain references to *Instances*, but rather maintains three sets of references: T, N, and U traces.

As introduced in our earlier work [38], we distinguish not just between the existence of a trace and its absence but instead use a trinary status: *T, N, U*. A *T-trace* describes an explicit decision by an engineer that there exists a trace between two artifacts. An *N-trace* describes a strong, explicit absence of a trace. Finally, a *U-trace* describes an uncertain trace, a situation in which an engineer has not made an explicit decision between a T- and an N-trace. This approach is compatible with other traditional works on traceability that only consider the absence or presence of a trace by treating the absence of a trace as a U-trace. We outline the benefit of T, N, and U traces in our prior work [38]–[40].

Programmatic *navigability* is one of the main focus points of our approach. *DesignSpace* supports seamless and uniform navigability through properties, where the property contains a reference to another instance. In the absence of traces, these referenced instances are typically artifacts from the same tool as set by the *Tool Connector*. As *Tool Connectors* are by definition tool specific, they are unaware of any cross-tool traces and hence cannot foresee in the artifact's *InstanceType* the possibility of any such a trace. Hence, we use the *DesignSpace*'s ability to dynamically alter an *InstanceType* to achieve *navigability* of trace links. Concretely, each of the possible endpoints for a trace matrix type (i.e., an *InstanceType*) is augmented with an additional property that points to its corresponding matrix row or column. In Figure 4, the creation of the *ReqToCodeMatrixType* resulted in a *traceToCode* property added in the *RequirementType* and a *traceToReq* property in the *C++Type*. Note that it is irrelevant which of the two *InstanceTypes* is managed via a row and which one via a column, as rows and columns have the same underlying datamodel and behave exactly identically.

Currently, we add trace type definitions to *DesignSpace* via its API programmatically, as we currently have not defined any external JSON or XML trace type definition format.

DesignSpace provides a folder-based storage system, so that the users can organize their artifacts in different folder hierarchies. Upon trace matrix instantiation, this storage mechanism is used for two aspects: first, to specify a storage location for the traces themselves, and second, to limit the trace endpoints to the set of artifacts placed in the specified folders. For example, in Figure 4, the trace matrix instance *TraceMatrixProject1* is initialized with folder *ReqFolderProj1* and *CodeFolderProj1*, thereby ignoring any *C++Type* instances and *RequirementType* instances in any other folder.

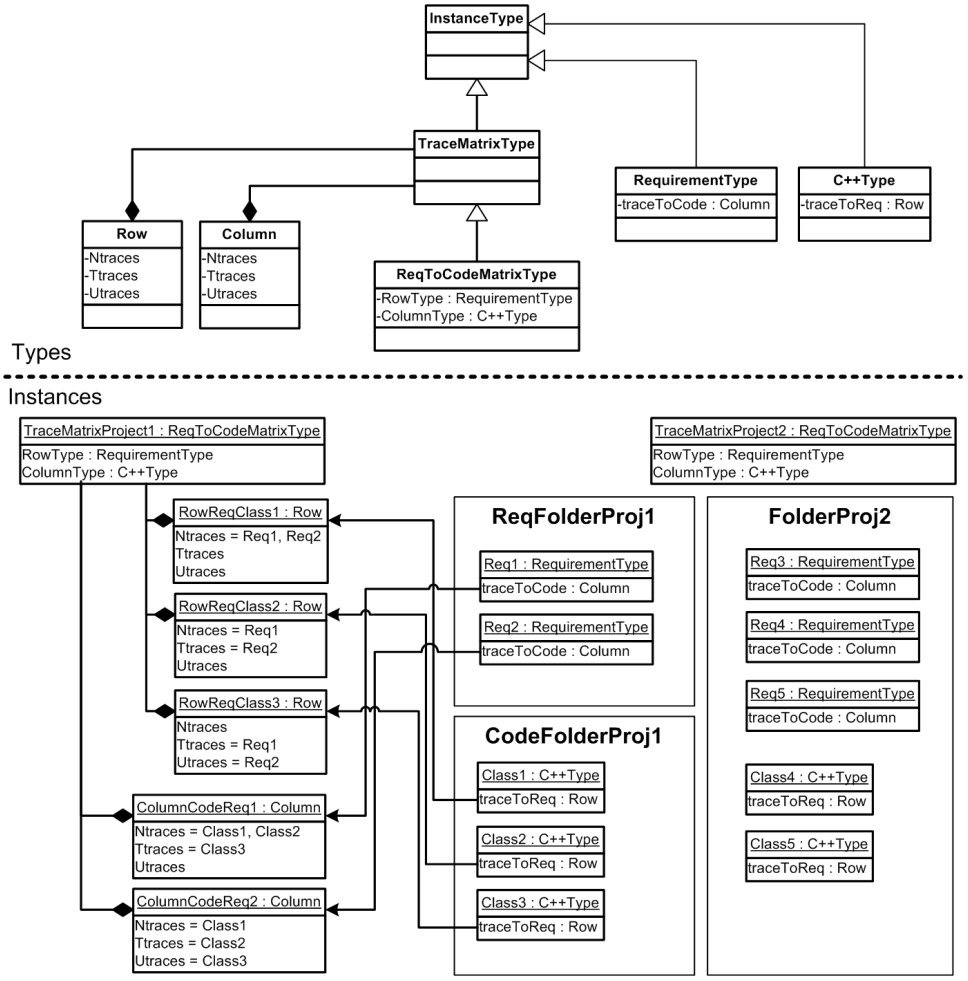

Fig. 4. Trace Meta Model and Instance examples (UML Class Diagram).

Upon the instantiation, every *RowType InstanceType* (here *RequirementType*) obtains a link to a *Column* instance having all *ColumnType* instances listed in the column's *Utraces* property. For example, in Figure 4 *Req1* and *Req2* point to their individual columns instances that represent the traces to the available classes *Class1*, *Class2*, and *Class3*. In Figure 4, we depict the traces in columns and rows as artifact identifiers and not as arrows back to instances for sake of clarity.

Likewise, every *ColumnType InstanceType* (here C++Type) obtains a link to a *Row* instance having all *RowType* instances listed in the row's *Utraces* property. The *Trace Matrix Maintainer* ensures that rows and columns are kept consistent.

Ultimately, this allows consuming traces from two perspectives. From the point of a trace matrix, one can generate the traditional matrix view and navigate onward to the individual artifact instances. More importantly, it becomes possible to traverse from an artifact instance to any other traced artifact instance via the trace properties. Obtaining all classes that a requirement *T-traces* to becomes simply a matter of navigating (in OCL syntax) "self.traceToCode.Ttraces".

D. Trace matrix growing and shrinking

As engineering work continues over time, engineers create new artifacts that are synchronized into *DesignSpace* via the *Tool Connectors* (cf. Figure3 (D)). The *Trace Matrix Maintainer* processes the creation events (cf. Figure3 (E)) and determines which trace matrices the new instance needs to be part of by matching the endpoint type of the trace matrix instances and the instance folder. For each matching trace matrix, all the instance traces are set to "undefined" (i.e., its row/column contains only *U-traces*), and all existing opposite endpoints obtain one corresponding new entry in their *U-trace* set (cf. Figure3 (F)). Likewise, removing instances results in the removal of its column/row and corresponding entries in the opposite endpoint trace sets.

E. Trace importing and creation

Traces across tools are often captured via IDs or URLs in dedicated fields that are typically just treated as plain strings in *Tool Connectors*. Recall that *Tool Connectors* are unaware of other tools and hence unable to insert any cross-tool links as they are unaware of the semantics of any properties used to define external trace links. Here, we apply

Link Translators that are aware of the two endpoint types for a particular trace they support, and are configured from which property to extract an artifact identifier, how to use that artifact identifier to resolve *DesignSpace* instance, and subsequently insert the respective entry in the trace matrix. Ultimately, creating the trace link requires only navigating to the respective trace property (e.g., *traceToCode* or *traceToReq*) of one of the trace's endpoint instances and adding the other endpoint instance to the *Ttraces* property. At this point, we provide a *Link Translator* for URL links. For example, if a GitHub issue is linked to a Jira issue, this can be specified with a certain encoding in the GitHub issue's description. When the description is stored within *DesignSpace* as part of the *Tool connector*'s synchronization, this encoding is automatically parsed, and the corresponding trace is established.

Aside from the navigable representation directly within the model structure, traces are available in matrix format from the trace matrix instance, which is used for visualizing in a table format in a graphical user interface, the *Trace Matrix Dashboard*. This is the main trace visualization available in our prototype at the moment, and supports the users in the process of *discovering trace endpoints* and creating new traces within *DesignSpace*. In the dashboard, the user selects the trace type and one corresponding trace matrix instance they are interested in. The matrix view is then populated with information from the trace matrix rows and columns instances.

Here, the user can set new traces by selecting the proper cell and changing its status to traced, marked with the symbol "T" or the description "Linked". Additionally, if there is knowledge that there is absolutely no trace between the cell's row and column instances, this can be marked as not traced, with the symbol "N" or the description "Not linked". For the purpose of supporting collaboration, as well as *extensibility*, any cell that is not specifically set to either traced or not traced, is marked with the symbol "U" as undefined.

F. Propagation trace changes back into tools

Once new or updated traces are available as a property directly from the artifact instance in *DesignSpace*, the task of synchronizing these changes back into the artifact originating tool arises. For intra-tool traces, the *Tool connectors* are able to conduct such a synchronization. For manually maintained cross-tool traces such as artifact IDs or URLs, the *Link Translators* would be able to translate the trace information back into the tool (cf. Figure3 (H)). For many artifacts such as the *C++Type* there might not exist a robust solution to embed trace information in the artifact within the tool at all. Here a tool plugin connecting to *DesignSpace* (cf. Figure3 (J)), such as the ones we describe in the next section, might provide a better solution for making use of traces. Overall, embedding cross-tool traces in their originating artifacts is a tool-specific task and cannot be solved in a generic manner.

In contrast to *Link Translators* or *Tool Connectors* that use a tool's API, *Trace Analysis Services and Plugins* typically require an extension of the engineering tool and hence are very application specific.

V. Preliminary Evaluation

The collaboration with our industry partners excludes disclosing in-depth software details and tool support that are part of their ongoing engineering processes. Therefore, our evaluation is based primarily on a scenario constructed to closely follow the processes and the artifacts involved in a product developed by an automotive company. In order to evaluate the feasibility of the proposed approach, we developed a prototype of our approach for the scenario, focusing on the traceability of requirements to code.

We use our prototype to trace requirements to the Java classes that implement them.[10] In this regard, the prototype is a simplified implementation of the features/properties described above (see Section II). This version does not consider the folder organization offered by *DesignSpace*, which implies each trace type can only be instantiated with one trace matrix that considers all the instances of each endpoint type, regardless of the location where they are stored. Considering this design decision in this scenario, the case study system includes three requirements (for sake of simplicity) and the Java source code of a robotic arm.

Our evaluation purpose is to document and create the corresponding trace links and, further, to exercise the navigability of the trace links by exploring automatic consistency checking. *DesignSpace* supports checking consistency rules through an integrated service, which allows the user to specify OCL constraints in the context of an instance type. Subsequently, the constraint will be checked for all the instances of said instance type, determining their consistency with regard to the constraint. Any newly created instance of the instance type will automatically be evaluated for its adherence to the constraint. Further, any change that happens to an instance automatically triggers the re-evaluation of the consistency rules.

We evaluate trace completeness by checking that each class in the source code is traced to at least one requirement. To obtain this, we first create a trace matrix that connects the requirement type to the Java class type representation in *DesignSpace*, naming the trace matrix "implements", as shown in Figure 5. We then introduce the traces into the matrix, with "Linked" representing a T-trace, and "Not linked" representing an N-trace. We leave a number of trace links undefined, as the relationship they represent has not been explored yet. The Point class is an example of Java class that does not have a link (i.e., neither traced nor not traced) to any requirement. Next, we access the consistency rule service and create a new consistency rule. This rule is attached to the Java class instance type and requires it to be linked to at least one requirement. After creating the rule, it is automatically checked in the background. When we navigate in the source code tool to one of the classes that are violating this constraint, such as the Point class mentioned before, a warning is informing us that a link to a requirement is missing, as shown in Figure 6.

[10] A video demonstration of this prototype is available at https://drive.google.com/file/d/1AgE78cItarPAWIgkf8wLH3BrOv0cbQrp/view?usp=share_link.

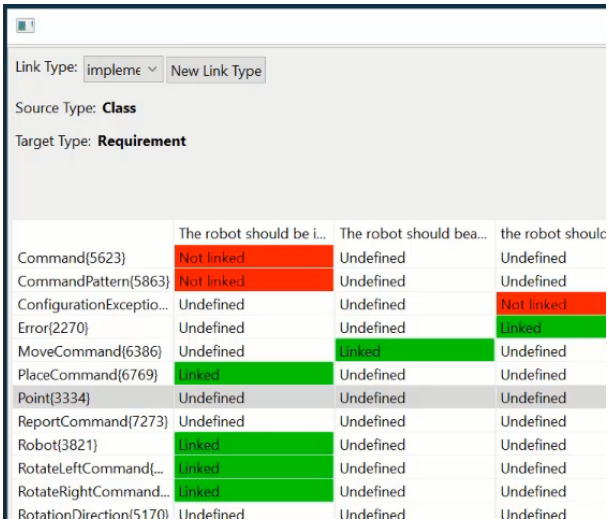

Fig. 5. Excerpt of artifacts and trace links in use at our industry partners.

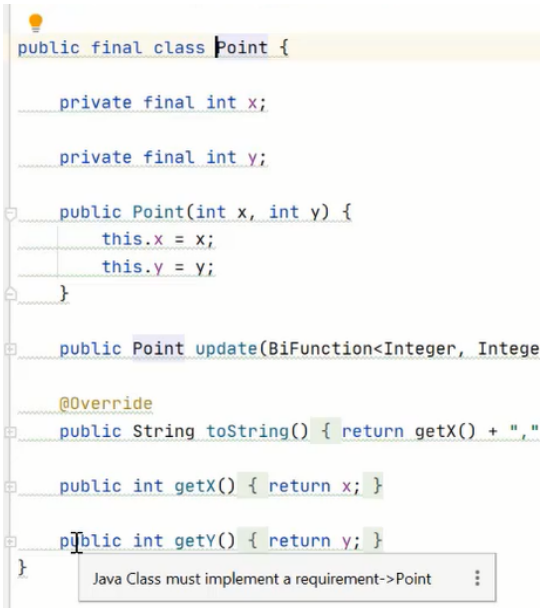

Fig. 6. Example of warning informing that a link to a requirement is missing.

This scenario highlights one of the many possible use cases in which the traces are an intermediary product to further automated engineering support. Besides consistency checking, different processes such as change impact analysis, test coverage reports, and identifying test cases for regression testing are some examples in which establishing and navigating traces across tools results in a significant increase in the efficiency and opportunity for automation. In the absence of navigable trace links, all these operations have to be done manually.

VI. Concluding Remarks

In this paper, we have discussed the remaining challenges in the face of supporting traceability in safety-critical and regulation-centric domains, especially in the automotive industry. In collaboration with our industry partners, we have identified a set of five desired properties for their current development process: discoverability, type checks, flexible definitions, navigability, and extensibility. These properties are still missing from the typical tool support offered to engineers for setting trace links, and can be mapped to open challenges in the literature. To fill this gap, we have proposed an approach that covers these characteristics to provide effective and actionable cross-tool traceability support. We have evaluated the feasibility of the approach in a scenario designed based on the processes followed by our industry partners.

There are a number of future development areas we are interested in further investigating. One such area is the issue of trace visualization. The trace matrix we offer has a number of advantages, from usability to offering a clear overview of the traces involved in a project. However, we have no certain indication whether it is the best visualization option for projects of all sizes and types. During our discussions with our industry partners, we have noted a few possible refinements or alternatives to this option, such as a filtered view per artifacts or process progress instead of the full matrix view. We consider further research necessary to determine what other visualization options are helpful in practice.

We also envisage that our approach can offer trace recommendations based on the relationships between artifacts and the trace information already set in the system (see Section IV). We plan to extend the context of these recommendations to be applicable to more use cases. We will also explore other trace recommendation systems studied in the literature, as well as different automation solutions, and how they could be integrated into *DesignSpace*.

Another area of interest we see as an opportunity for further investigation is integrating versioning into our traceability model. Currently, *DesignSpace* maintains a full and comprehensive history of the artifacts stored within it, but does not support the concept of versioning as used by our industry partners, i.e., maintaining multiple variants of the same artifact at the same time. Further research is necessary to understand how traceability can be best supported for such scenarios and how the traceability effort can be minimized for engineers.

Acknowledgment

This work has been supported by the Austrian Science Fund (FWF) grant P31989-N31 and P34805-N; the FFG Contract No. 881844: "Pro^{2}Future is funded within the Austrian COMET Program Competence Centers for Excellent Technologies under the auspices of the Austrian Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation and Technology, the Austrian Federal Ministry for Digital and Economic Affairs and of the Provinces of Upper Austria and Styria. COMET is managed by the Austrian Research Promotion Agency FFG"; the LIT Secure and Correct System Lab sponsored by the province of Upper Austria; and the COMET-K2 Center of the Linz Center of Mechatronics (LCM) funded by the Austrian federal government and the federal state of Upper Austria.

References

[1] M.-A. Peraldi-Frati and A. Albinet, "Requirement traceability in safety critical systems," in *1st Workshop on Critical Automotive Applications: Robustness & Safety*, ser. CARS '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 11–14.

[2] P. Mason, "On traceability for safety critical systems engineering," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, 2005, pp. 8 pp.–.

[3] J. Hill and S. Tilley, "Creating safety requirements traceability for assuring and recertifying legacy safety-critical systems," in *18th IEEE International Requirements Engineering Conference*, 2010, pp. 297–302.

[4] VDA QMC Working Group 13 / Automotive SIG, "Automotive SPICE Process Assessment / Reference Model," pp. 1–128.

[5] R. Messnarz, D. Ekert, T. Zehetner, and L. Aschbacher, "Experiences with aspice 3.1 and the vda automotive spice guidelines–using advanced assessment systems," in *European Conference on Software Process Improvement*. Springer, 2019, pp. 549–562.

[6] G. Macher, A. Much, A. Riel, R. Messnarz, and C. Kreiner, "Automotive spice, safety and cybersecurity integration," in *Computer Safety, Reliability, and Security*, S. Tonetta, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2017, pp. 273–285.

[7] E. Edwar, S. Sameh, and I. Sobh, "Aspice applicability on new automotive technologies (ai)," in *Systems, Software and Services Process Improvement*, M. Yilmaz, P. Clarke, R. Messnarz, and B. Wöran, Eds. Cham: Springer International Publishing, 2022, pp. 430–440.

[8] B. Brosgol and C. Comar, "Do-178c: A new standard for software safety certification," ADA CORE TECHNOLOGIES NEW YORK NY, Tech. Rep., 2010.

[9] G. Islam and T. Storer, "A case study of agile software development for safety-critical systems projects," *Reliability Engineering & System Safety*, vol. 200, p. 106954, 2020.

[10] J. A. McDermid, "Issues in developing software for safety critical systems," *Reliability Engineering & System Safety*, vol. 32, no. 1-2, pp. 1–24, 1991.

[11] R. E. Lopez-Herrejon, J. Martinez, W. K. G. Assunção, T. Ziadi, M. Acher, and S. Vergilio, *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Springer Nature, 2022.

[12] R. Capilla, J. Bosch, K.-C. Kang *et al.*, "Systems and software variability management," *Concepts Tools and Experiences*, vol. 10, p. 2517766, 2013.

[13] R. Maschotta, M. Hammer, T. Jungebloud, M. Khan, and A. Zimmermann, "Model-driven aspect-specific systems engineering in the automotive domain," in *IEEE International Conference on Recent Advances in Systems Science and Engineering (RASSE)*, 2021, pp. 1–8.

[14] O. Gotel, J. Cleland-Huang, J. H. Hayes, A. Zisman, A. Egyed, P. Grünbacher, A. Dekhtyar, G. Antoniol, J. Maletic, and P. Mäder, "Traceability fundamentals," *Software and systems traceability*, pp. 3–22, 2012.

[15] S. Demi, M. Sanchez-Gordon, and R. Colomo-Palacios, "What have we learnt from the challenges of (semi-) automated requirements traceability? a discussion on blockchain applicability," *IET Software*, vol. 15, no. 6, pp. 391–411, 2021.

[16] S. Maro, J.-P. Steghöfer, and M. Staron, "Software traceability in the automotive domain: Challenges and solutions," *Journal of Systems and Software*, vol. 141, pp. 85–110, 2018.

[17] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[18] A. Demuth, R. Kretschmer, A. Egyed, and D. Maes, "Introducing traceability and consistency checking for change impact analysis across engineering tools in an automation solution company: An experience report," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 529–538.

[19] L. Marchezan, W. K. G. Assunção, E. Herac, F. Keplinger, A. Egyed, and C. Lauwerys, "Fulfilling industrial needs for consistency among engineering artifacts," in *45th International Conference on Software Engineering (ICSE) - Software Engineering in Practice*, 2023, pp. 1–12.

[20] L. Marchezan, W. K. G. Assuncao, R. Kretschmer, and A. Egyed, "Change-oriented repair propagation," in *International Conference on Software and System Processes and International Conference on Global Software Engineering*. ACM, 2022, pp. 82–92.

[21] L. Marchezan, R. Kretschmer, W. K. G. Assunção, A. Reder, and A. Egyed, "Generating repairs for inconsistent models," *Software and Systems Modeling*, vol. 22, no. 1, pp. 297–329, apr 2022.

[22] C. C. Raţiu, W. K. G. Assunção, R. Haas, and A. Egyed, "Reactive links across multi-domain engineering models," in *25th International Conference on Model Driven Engineering Languages and Systems*. ACM, 2022, pp. 76–86.

[23] A. Demuth, M. Riedl-Ehrenleitner, R. E. Lopez-Herrejon, and A. Egyed, "Co-evolution of metamodels and models through consistent change propagation," *Journal of Systems and Software*, vol. 111, pp. 281–297, 2016.

[24] W. D. F. Mendonça, W. K. G. Assunção, and S. R. Vergilio, "Feature-oriented test case selection during evolution of high-configurable systems," in *27th Systems and Software Product Line Conference (SPLC)*. ACM, 2023, pp. 1–11.

[25] W. D. F. Mendonça, S. R. Vergilio, G. K. Michelon, A. Egyed, and W. K. G. Assunção, "Test2feature: Feature-based test traceability tool for highly configurable software," in *26th ACM International Systems and Software Product Line Conference - Volume B*. ACM, 2022, pp. 62–65.

[26] J. Cleland-Huang, A. Agrawal, M. Vierhauser, and C. Mayr-Dorn, "Visualizing change in agile safety-critical systems," *IEEE Softw.*, vol. 38, no. 3, pp. 43–51, 2021.

[27] F. Tian, T. Wang, P. Liang, C. Wang, A. A. Khan, and M. A. Babar, "The impact of traceability on software maintenance and evolution: A mapping study," *Journal of Software: Evolution and Process*, vol. 33, no. 10, p. e2374, 2021.

[28] D. Fucci, E. Alégroth, and T. Axelsson, "When traceability goes awry: an industrial experience report," *arXiv preprint arXiv:2206.04462*, 2022.

[29] L. Westfall, "Bidirectional requirements traceability," *White Paper, The Westfall Team, Dallas*, 2006.

[30] M. Shahid, S. Ibrahim, and M. N. Mahrin, "An evaluation of requirements management and traceability tools," *World Academy of Science, Engineering and Technology, WASET*, vol. 1, no. 1, pp. 1–6, 2011.

[31] D. Amalfitano, V. De Simone, R. R. Maietta, S. Scala, and A. R. Fasolino, "Using tool integration for improving traceability management testing processes: An automotive industrial experience," *Journal of Software: Evolution and Process*, vol. 31, no. 6, p. e2171, 2019.

[32] R. Wohlrab, J.-P. Steghöfer, E. Knauss, S. Maro, and A. Anjorin, "Collaborative traceability management: Challenges and opportunities," in *2016 IEEE 24th International Requirements Engineering Conference (RE)*. IEEE, 2016, pp. 216–225.

[33] H. Tufail, M. F. Masood, B. Zeb, F. Azam, and M. W. Anwar, "A systematic review of requirement traceability techniques and tools," in *2nd international conference on system reliability and safety (ICSRS)*. IEEE, 2017, pp. 450–454.

[34] S. Maro, J.-P. Steghofer, E. Knauss, J. Horkoff, R. Kasauli, R. Wohlrab, J. L. Korsgaard, F. Wartenberg, N. J. Strøm, and R. Alexandersson, "Managing traceability information models: Not such a simple task after all?" *IEEE Software*, vol. 38, no. 5, pp. 101–109, 2020.

[35] J. M. C. de Gea, C. Ebert, M. Hosni, A. Vizcaíno, J. Nicolás, and J. L. Fernández-Alemán, "Requirements engineering tools: An evaluation," *IEEE Software*, vol. 38, no. 3, pp. 17–24, 2021.

[36] M. Gatrell, "The value of a single solution for end-to-end alm tool support," *IEEE Software*, vol. 33, no. 5, pp. 103–105, 2016.

[37] A. Demuth, M. Riedl-Ehrenleitner, A. Nöhrer, P. Hehenberger, K. Zeman, and A. Egyed, "Designspace: An infrastructure for multi-user/multi-tool engineering," in *30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1486–1491.

[38] A. Ghabi and A. Egyed, "Code patterns for automatically validating requirements-to-code traces," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2012, pp. 200–209.

[39] H. Kuang, P. Mäder, H. Hu, A. Ghabi, L. Huang, J. Lü, and A. Egyed, "Can method data dependencies support the assessment of traceability between requirements and source code?" *J. Softw. Evol. Process.*, vol. 27, no. 11, pp. 838–866, 2015. [Online]. Available: https://doi.org/10.1002/smr.1736

[40] M. Hammoudi, C. Mayr-Dorn, A. Mashkoor, and A. Egyed, "Tracerefiner: An automated technique for refining coarse-grained requirement-to-class traces," in *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*. IEEE, 2021, pp. 12–21. [Online]. Available: https://doi.org/10.1109/APSEC53868.2021.00009