(Not) Yet Another Metamodel For Traceability

Edouard R. Batot[1], Sebastien Gérard[2], and Jordi Cabot[1,3]

[1] Universitat Oberta de Catalunya – {ebatot,jcabot}@uoc.edu
[2] CEA LIST – sebastien.gerard@cea.fr
[3] ICREA – jordi.cabot@icrea.cat

Abstract. Traceability helps explaining the execution and evolution of software systems and it is a key input in many software engineering tasks such as program understanding, maintenance and debugging. Several metamodels to facilitate the representation of traces and links between related artefacts have been proposed. There exists a plethora of approaches that focus on distinct segments of the software development processes and products. Nevertheless, we claim they lack the mechanisms to express important traceability aspects such as the quality of traces, their gradual decay, and the evidences supporting them. This affects the benefits traceability can bring to the above-mentioned tasks. This paper presents a more expressive traceability metamodel, covering all the missing dimensions in a single, but extensible and modular, design. This modularity facilitates the integration of our solution in other modeling languages or its partial adoption when only some specific traceability aspects are needed. Its extensibility facilitates its customization (*e.g.,* in terms of the types of links and artefacts) to better cover specific domains.

Keywords: Software Engineering, Model-Driven Development, Traceability, Metamodeling

1 Introduction

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 [25] as 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.

The need for traceability has always been a recurrent aspect of software 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 traces offer a different perspective on a system, arbitrary and customizable, where the relationships between elements is the most salient artefact. Traceability rises awareness on specific purposes or goals [12] and has been proven useful in a diverse number of software engineering challenges [21]. It is transient to any software maintenance effort such as change impact prediction [24,20], debugging [29,1], feature location [13,34] or certification [36] among many others [27].

Importance of traceability is reflected in the production of many metamodels targeting the modeling of traceability aspects. Many of them focus on specific aspects or domains where the traceability mechanism is applied [3,47,46]. Metamodels flourish but their knowledge remains scattered among software engineering research fields. There is even work dedicated to the engineering of metamodels for traceability that offers a language dedicated to defining traceability metamodels [14]. Overall, we are still missing a generic metamodel for traceability that covers not only the representation of artefacts, traces and links between them but also quality aspects that can be used to interpret the relevance and integrity of traces.

The surge of artificial intelligence (AI) applications in software engineering makes good traceability support even more important as part of explainability mechanisms inherent to these AI applications [38,35]. And conversely, AI can also be a mechanism to infer new traces among sets of artefacts [6,21], which will need to come hand in hand with the proper explainability support, specially considering the non-deterministic nature of information retrieval algorithms. This non-deterministic nature draws in a significant degree of uncertainty about the results such algorithms may yield. Traces automatically identified show variable confidence. This dimension should be considered as a core concern to traceability.

In this sense, the contribution of this paper is the definition of Trace*a*: a generic and extensible traceability metamodel integrating quality concerns (e.g. decay, confidence and explainability) in the definition of traces. The design of the metamodel favours also its adaptation to specific application domains and model-driven toolchains to open the door to a new generation of techniques (E.g., for impact analysis) that could benefit from our more expressive metamodel.

The rest of this paper is structured as follows. Section 2 introduces the state of the art through a comparison between a selected set of approaches addressing the modeling of traceability. Section 3 presents the limitations of current solutions through their main quality concerns. In Section 4, we show and depict our metamodel Trace*a* and display an illustrative example. We discuss about the integration of Tracea and more generally of traceability modeling into existing tooling in Section 5 before we conclude in Section 6.

2 State of the art

We have identified over 80 approaches aimed a modeling traces and tracing activities [5]. We describe in this section a selection of the most representative publications and then summarize their main limitations as the key challenges our solution will aim to overcome.

Most works on modelling traceability come, historically, from the requirement modelling community. Traces are seen as "links" from requirements to their (sub)components, and to their design and/or implementation artefacts [44,22]. Specially relevant is the work by Goknil *et al.* [19] that includes a metamodel for traces, mechanisms for consistency checking and inferencing, and tooling for change impact analysis.

In the model-driven engineering (MDE) community, research work can be classified into proposals focusing on modeling the heterogeneity of artefacts – with numerous contributions aiming at linking text artefacts to design models [41], or addressing the entanglement of (non functional) requirements [48] – and proposals adapting traceability to specific application areas such as the automotive and robotic industry [15,40,42].

In this group, we see several publications that include *custom* metamodels built *ad hoc* to solve specific model transformation issues [30,28].

Other modeling approaches are aimed at establishing an automated trace generation process, e.g. for requirement traceability. For example, Spanoudakis *et al.* We see these cases as sidesteps from traceability modeling since the works aim at generating traces (that need being modelled) rather than modelling traces (that need being generated). As a consequences, the presented metamodels are specific to the types models source or target of the generation (*e.g.,* BPMN models [39], or data warehouse models [33]). Natural language is also often used in this type of automated processes. In this case, approaches target the extraction of semantics (or *meaning)* from textual requirements. These publications model text blocks with their dependencies and the dependencies to specific third party artefacts (*e.g.,* for MDE: [41], for AADL [45], for agile user stories [10]).

Recently, researchers attempted more generic approaches to traceability, closer to our own goal. Building on previous knowledge in specific domains, authors describe their attempts to synthesise traceability requirements. For example, Azavedo et al. [4] created a metamodel with explicit (57) relationship types and (12) different kinds of artefacts based on an arbitrary separation of software development tasks (*e.g.,* Implementation, Verification, Modification, Homologation). On the other hand, Heisig *et al.* present a modeling approach to traceability that includes both a basic metamodel with a plugin mechanism (using XText) that allows user to define their specific representations for links and artefacts [23].

While these latter approaches do represent in advance in the generalibility and adaptability of traceability metamodels, our approach offers a higher granularity and decomposition while integrating several quality concerns (decay, confidence, and explainability). Table 1 summarizes existing works regarding these core traceability aspects. As shown in the table, most publications consider a single trace level, which limits the complexity and diversity of problems where traceability can be applied. There is also a significant lack of consideration for quality aspects. Consistency is merely mentioned and confidence is strictly forgotten – none of the selected approaches mention it. Explainability is reported in a few cases but remains scarce and no common appreciation has emerged yet.

3 Requirements

Following up on the state of the art analysis, this section details the traceability requirements we believe are needed in order to have a complete traceability modeling solution, able to be used in a variety of scenarios, including industrial applications. Next section describes our proposed metamodel and how its different components satisfy these requirements.

3.1 Adaptability & Configurability

Reusability of a traceability solution is key for its industrial adoption. When traces are seen as useful only to conform a very specific requirement (e.g. software certification deadline). Enterprises have shown that it is *easier* or *cheaper* to execute it as a manual and *ad hoc* process [12].

Approaches\Quality	**Adaptability**	**Granularity**	**Consistency**	**Confidence**	**Explainability**
Goknil *et al.*[19]	Generic types	1-step links	–	–	–
Taromirad *et al.*[44]	Fixed types	1-step links	–	–	–
Haidrar *et al.*[22]	Fixed types	1-step links	Timeliness	–	–
Sannier *et al.*[41]	Specific types	1-step links	–	–	–
Dubois *et al.*[15]	Specific types	1-step links	–	–	–
Sanchez *et al.*[40]	Specific types	Multi steps	–	–	Evidences
Yrjonen *et al.*[48]	Specific types	Multi steps	Timeliness	–	Evidences
Jimenez *et al.*[28]	Specific types	Multi steps	(not applicable)	–	–
Levendovsky *et al.*[30]	Generic types	1-step links	Context sensitive	–	Evidences
Wang *et al.*[45]	Fixed types	1-step links	–	–	–
Carniel *et al.*[10]	Fixed types	Multi steps	–	–	–
Spanoudakis *et al.*[43]	Generic types			–	–
Pavalkis *et al.*[39]	Specific types	1-step links	–	–	Agent
Maté *et al.*[33]	Specific types	1-step links	–	–	–
Azevedo *et al.*[4]	Generic types	Multi steps	Timeliness	–	–
HeiSig *et al.*[23]	Generic types	Compositional	Context sensitive	–	–

Table 1. Occurrences of the main properties for modeling traceability.

We aim for a metamodel that it is configurable and adaptable to maximize its reusability in a number of application scenarios thus favouring its adoption by companies. For example, a specific certification paragraph might be better suited (*i.e.,* more precise) for the user than the entire certification document containing this paragraph. Or, if the purpose is to trace the impact of changes in a model on the source code, does the user want to know about the *occurrence* or the *location* of a change? Does the user want to find the right *file* or the right *class*, the right *method*, or the right *package*? In other words, *what kind of artefacts* of the software product is of interest (*e.g.,* design models, source code) and to *which level of granularity* ?

High-level types for artefacts as well as peculiar level of granularity must be adequately designed in the tracing solution.

Configurable tracing A trace is commonly expressed as the combination of atomic trace links representing direct connections between a number of artefacts. For example, a certification document (*e.g.,* ISO-26262 [26]) is "linked" or "related" to a set of design documents, or models, themselves being used for (or "relating to") the generation of source code or other related artefacts such as behavioral models [31]. Depending on authors intentions and problem constraints, they define traces with a single or multiple sources and end with one or many targets.

There is little attention put on more complex tracing purposes such as the complete sequence from certification specifications to source code implementation, or long reach tracing ability involving sequences of artefacts or decisions in chain.

We believe a traceability metamodel must come with several levels of granularity to enable users express traceability relationships either at a coarse-grained or a fine-grained level depending on their needs. Moreover, if defined at a fine-grained level, the model should be able to use that information to propagate those trace links to the container components to offer automatically the coarse-level view as well.

Adaptable tracing The exact set of artefact types we must trace in a project may not be completely known upfront. And they will probably change over time. As such, we need our metamodel to be extensible with new artefact types that perfectly match the elements users want under scrutiny. To ensure reuse, it is convenient to craft a base metamodel and adapt it to specific situations where traceability will be needed.

It is important not only to be able to extend this base metamodel with new artefact types but also with new types of relationships (each one with its own semantics) among them. Most approaches offer fixed types among which to choose. They are either specific to the domain of application [39,17] or generic and relate to a greater extend to the nature of the artefacts [42,16]. As Maro *et al.* warn, "avoid implicit, convention-based traceability links and strive instead for explicit links that can be checked with tool support," [32]. in that regard, a distinction based on the nature of artefacts to provide high level types is recommended by many traceability researchers [12].

3.2 Consistency

One of the main argument against investing in automated traceability support remains the cost of maintaining traces up-to-date [12]. Software systems evolve and endure maintenance bug fixing and patches that can potentially modify their constitutive elements at every level. Even their architecture changes through time to cope with increasing scalability needs, to comply with new privacy regulations, or to add or modify the panel of features offered to the different kind of "users" of the system. Tracing is no alien to this phenomenon and the cost to maintain traces consistent with the system increases hand in hand with the system volatility.

There is no consensus on the means to ensure that traces remain *consistent* to the system. Yet, the naive method that consist in rebuilding the entire graph of traces each time *from scratch* does not scale [42]. Gervasi *et al.* exploit the information contained in previously defined traces, in order to facilitate the creation and ongoing maintenance of traces, as the requirements evolve [18]. Seibel *et al.* have shown that the MDE paradigm offers auspicious horizon to the maintenance of traces [42]. They execute rules in order to maintain a set of links *representative* to the trace types predefined beforehand. Authors extend the concept of *timestamp* to consider context changes and thus to reflect better the system volatility.

We agree that traceability metamodels should be able to represent temporal information [8], not only for the traces but for all the traced elements so that we can compare them and evaluate the potential traces decay.

3.3 Confidence

Decay is not the only factor that can affect our confidence on the consistency and relevance of a trace. The execution of automated processes to identify traces raises *uncertainty* about the actual existence of the results they yield. Learning techniques, using deep learning algorithms such as in [21], offer to bridge the cognitive gap among artefacts of different nature but accuracy is never perfect. There lies an open topic at the intersection between "traditional" and "AI-enabled" software practices. Systems with AI-enabled components [generally probabilistic] can have a high margin of error due to the uncertainty that often follows predictive algorithms [38]. Taking account of the non deterministic nature of AI modules is a key factor for AI-enabled software of quality. Nevertheless, even a manual trace identification process can have some uncertainty as designers may not be completely sure about the real relationship between components that may have been created a long time ago.

Therefore, we need to be able to express in a traceability model the confidence we have on the traces. Where to draw the line between a useful trace based on this confidence level depends on the envisioned application. There is a trade-off to evaluate between the level of confidence and the level of critically of the project. If the purpose is to evaluate a requirement change impact on the source code, traces with a low confidence level may trigger false positive and generate some additional work but still be reasonable useful. If traces are used as part of a security certification, a high confidence level is a must to obtain valid results. The propagation of uncertainty is an open topic that authors attempt to address with a mix of boolean logic and Gaussian statistics [7] and that should be applied to trace uncertainty as well.

3.4 Explainability

Explainability is more and more important in any software system due to increasing transparency, ethical and regulatory concerns. Users do not only require that the answer of the system is the one expected, they need to know *how* did the system proceeded to yield such answer. AI-enabled systems raises this issue to a new level of salience.

To support explainability, we need to be able to explain the reason why a link among artefacts is identified. Trace links can be elicited manually (and a textual report would precise why in natural language); or identified automatically using information retrieval and rule-based techniques. We need to be able to register this nature as well as the details of the identification process.

For example, programmers often use mnemonics for identifiers that help associate code with high-level concepts in the requirements and vice-versa [2]. If traces, used for quality audit, have been identified thanks to a rule-based approach exploiting this mnemonics, this information needs to be part of the traceability model as those traces could be later used as evidences to automatically check for potential mismatches or coverage analysis of requirements not supported in the implementation.

4 Trace*a* Metamodel

Based on the previous analysis, we present in this section a new traceability metamodel, called Trace*a*. We aim not to present yet another traceability metamodel but one that learns from and supersedes previous proposals in order to provide a more complete metamodel that also responds to the requirements made explicit in Section 3. To foster legibility, we have added a simple but complete example of the use of our metamodel to conclude this section.

4.1 Adaptable and configurable traces

In Trace*a*, we start from the common core trace representation in several of the existing metamodels but expand and refine it with a more configurable structure that allows one to define the proper level of granularity.

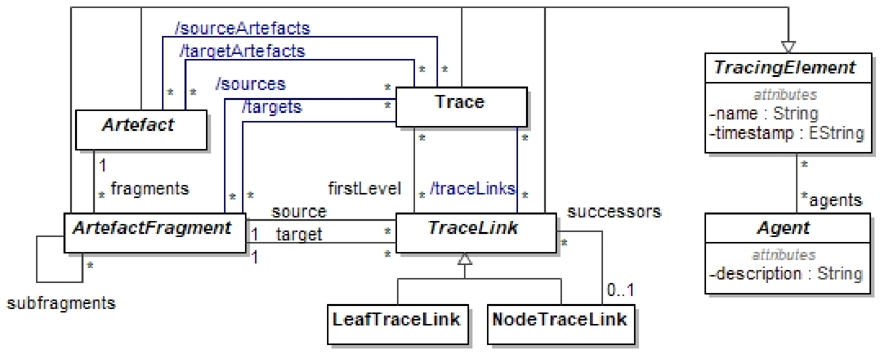


Fig. 1. Core compositional nature of the Tracea

Fine grain tracing structure. The excerpt in Fig. 1 describes the composition scheme of links into a forest-like structure. Its atomic elements are `TraceLinks`. A trace link refers to a source and a target `ArtefactFragment`(see below). It may be a leaf – which means it has no successor links ; or a node – which means the trace does not *end* with this link. A `Tracelink` is a composite of `LeafTraceLink` and `LeafTraceNode`. A `Trace` starts from a set of trace links "`firstLevel`" that connect to their respective trees. The set of derived `TraceLinks` from the transitive closure of `firstLevel` is contained in the reference `traceLinks`. In the same manner, the set of source and target artefacts and fragments of traces (see blue references in Fig. 1) is derived as well (see Listing 1.1). `Trace` and `TraceLink` are subtypes of `TracingElement` with a unique identifier (name), a timestamp to address consistency issues (see below), and one or more `Agent` related.

```
context Trace inv firstLevels:
   self.traceLinks
   ->includesAll(self.firstLevels)

context Trace inv sources:
   self.firstLevels
   ->collect(source)
   ->includesAll(self.sources)

context Trace inv targets:
   self.tracelinks
   ->collect(target)
   ->includesAll(self.targets)
```

```
context Trace inv sourceArtefacts:
   self.firstLevels
   ->collect(source)
   ->includesAll(
      self.sourceArtefacts.fragments)

context Trace inv targetArtefacts:
   self.tracelinks
   ->collect(target)
   ->includesAll(
      self.targetArtefacts.fragments)
```

Listing 1.1. OCL expressions

Adaptable artefacts and relationships. Existing traceability works differ a lot in the *kind* of artefacts they target. A unified ontology of traced artefacts has yet emerged but a common approach is to distinguish between the nature of artefacts. *I.e.,* from *text intensive* (*e.g.,* requirements, certification) to *structure intensive* (*e.g.,* source code, test cases). In Trace*a*, we specialize `Artefact` with `TextArtefact`, `ModelArtefact`, `CodeArtefact`, and `TestArtefact`. The list is not exhaustive. These high-level types support the user in defining her own (sub)types. They are anchors to refine artefacts and their fragments at an adequate level of granularity.

To freely adjust the granularity of the artefacts under scrutiny, Trace*a* suggests the fragmentation of `Artefact`. An `ArtefactFragment` defines a part of an artefact that is of interest (*e.g.,* a method in a class; a section in a text document). Trace*a* implements a high-level separation for artefacts and relationships. This enables the separate customization of artefact types and the semantic relationships among them. The typing of relationships is versatile and we distinguish two kinds depending on the domain they apply to: `DomainType` and `EngineeringType`. In the former, the semantic of the final user (*i.e.,* its domain of application) is targeted. In the latter, the concepts used by engineers or modelers are targeted (the domain of engineering). Same as for the artefact types, relationship types are also expected to be customized [37] when needed.

Fig. 2 shows an excerpt of the Trace*a* metamodel that focus on aspects related to the adaptability of our approach.

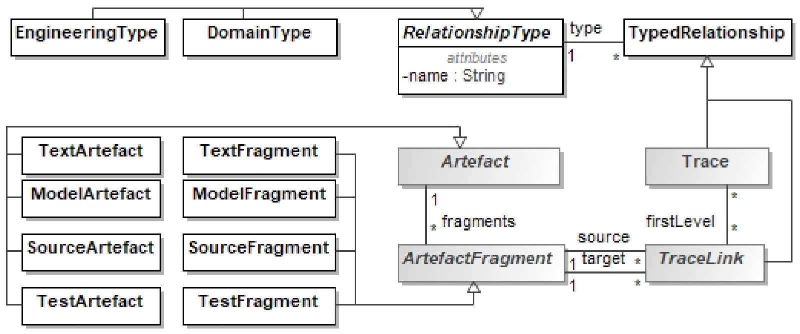


Fig. 2. Customization of artefacts and relationships in Tracea

4.2 Confidence of trace links

In Trace*a*, the confidence of a trace (`TraceConfidence`) or of a trace link (`Link-Confidence`) is a statement with a real number value representing the level (from 0 to 1) at which the trace is certain to exist in the system. It is a statement about the relevance of a trace. A trace and its links are made by `Agents`. As illustrated in Fig. 3, an agent may be human (*e.g.,* when traces are elicited manually), or an agent may be a machine (*e.g.,* when an algorithm identify the trace automatically).

4.3 Trace Consistency

To address the issue of gradual decay, tracing elements must be considered alike with other software artefacts. Their evolution must be scrupulously and synchronously monitored. To be able to represent (and later reason) on the potential decay, we add a timestamp attribute to `TracingElement` (see Figure 3), *de facto* transforming all metaclasses inheriting from it in temporal elements. We can then use these timestamps to compare the age of a trace compared to the age of the elements traced by it and, if needed, update the confidence we have in that trace, or trace link, accordingly.

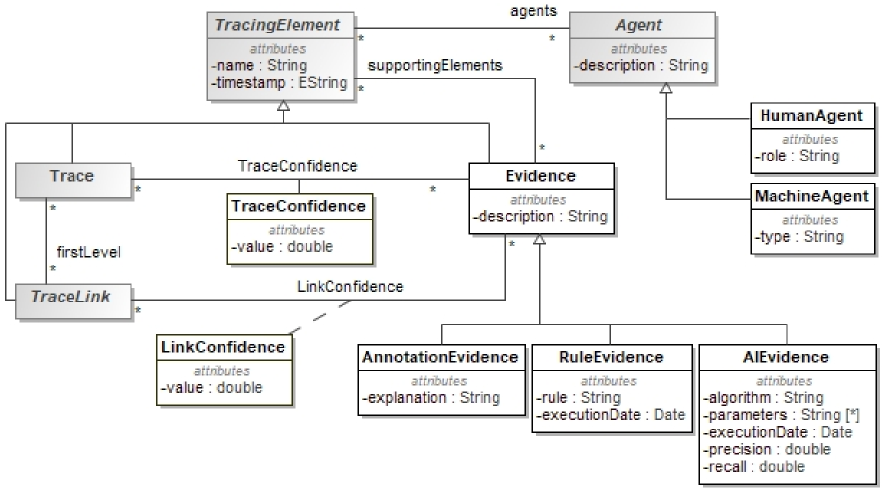


Fig. 3. Representing Explaining traceability with confidence value, agents and evidences

4.4 Explainability

Traces are a key element in many software engineering activities. Therefore, engineers may not want to just take them at *face value* but ask for explanations on how and when the trace was created. Previous subsection covered the *when*, here we focus on the *how*.

The degree of confidence may be justified with evidences, if they exist, to explain the rationale behind the quantitative value. In case of links automatically identified, an evidence instance can record the information necessary to reproduce the identification process or at least to partially explain it.

More precisely, as can be seen in Fig. 3, an evidence refines into three sub-types: `AnnotationEvidence` contains a textual description ; `RuleEvidence` contains a rule (or a set of rules) in a textual field, as well as the execution date ; and `AIEvidence` contains attributes that will help reproduce the learning scenario, *e.g.,* the kind of algorithm, a reference to the training set and the associated precision and recall the algorithm, and others parameters. An `Evidence` explanation can also point to other supporting tracing elements. These elements testify, or illustrate the evidence and will be useful for later consistency check.

Every evidence is also optionally endorsed by a set of Human or Machine agents, which further helps in the explainability of the trace beyond the description and attribute values stored in the Evidence object itself.

4.5 Illustrative example

In this section, we introduce a simple illustrative example: tracing the impact of a change in the requirements onto their implementation in Java classes. We show through this example the customization of artefacts and relationships, the importance of a quality evaluation, and how we circumvent consequences of using AI-enabled modules for trace identification. In this example, links are relating requirements to Java classes that undergo modifications.

Customization of artefacts and relationships In this case, a trace aligns two kinds of artefacts: `Requirement specification` and `Source class`.

These artefacts are too complex to be used at a coarse level of granularity. Java classes may comprise hundreds (or even thousands) of lines of code, requirement specification documents contain hundreds of sections. To address this size issue, a source `Artefact` (e.g., a class) is decomposed into smaller part (such as methods). In the same manner, specification documents are decomposed into sections. Listing 1.2 shows an excerpt of our textual concrete syntax applied to this example where we can see the fragmentation of artefacts. The structure of the traced system is first described with artefacts and fragments sections. For legibility concern, the only kind of relationship in that example is `Implement`, *i.e.,* a source class *implements* a requirement section.

```
artefacts {
   Requirement r_01 {fragments {sAuth
     , sLogout}},
   Source Login.java {fragments {
     mLogin, mLogError, mLogout }},
}
fragments {
   RequirementSection sAuth { },
   RequirementSection sLogout { },
   Method mLogin { },
   Method mLogError { },
   Method mLogOut { },
}
relationshiptypes {
   EngineeringType Implements {},
}
agents {
   HumanAgent 5e8a5T1e4,
   MachineAgent Rd15OUA5RD
}
```

Listing 1.2. Artefacts, Fragments, Relationships, and Agents declaration

```
Trace ChangeImpact {
   tracelinks {
      NodeTraceLink link01 {
         source sAuth
         target mLogin
         successors {link02}
         relationshiptype Implements
         agents 5e8a5T1e4
      },
      LeafTraceLink link02 {
         source mLogin
         target mLogError
         relationshiptype Implements
      },
      LeafTraceLink link03 {
         source sLogout
         target mLogout
         successors {}
         relationshiptype Implements
         agents Rd15OUA5RD
         confidence c01
      },
   }
}
```

Listing 1.3. Trace instance

The concrete traces are recorded as illustrated in Listing 1.3. A Trace is identifiable by its name and contains trace links whose composition is described through successors. This example is the minimalist expression of a trace. Each and every element is susceptible to refer to an `Agent` that indicates who and what is the nature of that “who” responsible for the edification of the trace. In our case, `link_01` and `link_03` have referees.

Explainability for AI-enabled traceability There exists automated evaluation techniques for change impact that predicts which classes are most likely to change. In this scenario, a change in the requirements links to *potentially* impacted classes, and to *actually* modified classes. This distinction shows a distinction in nature of the links themselves. The former is more inclined to suffer a low level of confidence than the automatized latter. In our case, Listing 1.3 shows that links `link_01` and `link_02` have been manually identified, and thus, there the confidence is `1.0` whereas `Link_03` has been automatically suggested and boasts a confidence of `0.8`. This level of confidence relies on evidence based on the algorithm employed, its parameterization, and its training setting. As can be seen in Listing 1.4, a confidence is related to a `Trace` or a `TraceLink` and a set of `Evidences`.

Evidences may be an `AIEvidence` like the one we just described, or `RuleEvidence` that relates patterns used for automatic identification, or `AnnotationEvidence` that simply contains a textual explanation of the evidence. In the example in Listing 1.4, the confidence `value` represents the confidence in the prediction that the method *mLogin* is impacted by a change in requirement `sAuth`. This prediction has been made using a specific algorithm which run settings can be found in the evidences section.

```
confidences {
   Confidence c01 {
      value 0.8
      evidence {Evidence_link03}
   },
}
evidences {
   AIEvidence Evidence_link03 {
      algorithmUsed "AI4All"
      parameters {"platform:/resource/training/pos_202012"}
      executionDate "20201207-123536"
      trainingResults .8 .7
      impactedElements ("link02", mLogin, otherMethod)
   },
}
agency {
   Rd15OUA5RD {Evidence_link03}
}
```

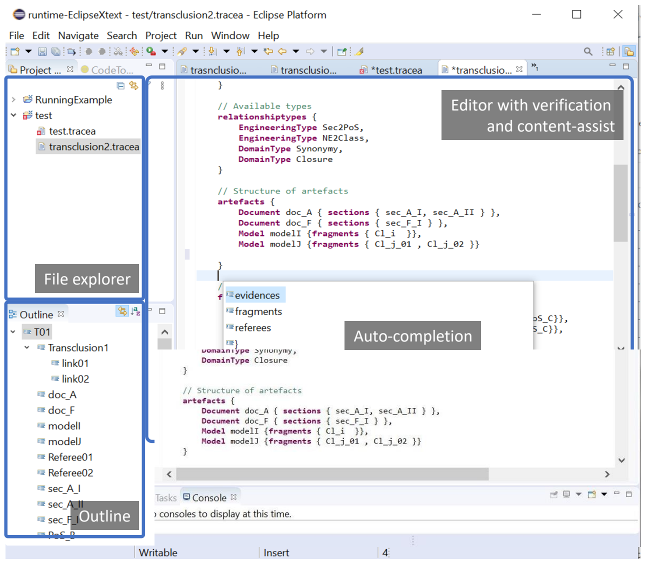
Listing 1.4. Confidence, evidence, and agency

5 Integration and Tool Support

An Xtext-based[4] definition of our metamodel is available on a Git repository[5]. As concrete syntax, we are using the JSON textual syntax shown in the examples above and illustrated in Fig. 4.

Beyond this option, we have also integrated Trace*a* on top of Capra[6]. Eclipse Capra is a traceability management tool offering some interesting features to edit and visualize traces, including traceability matrices and graph visualisations. Capra includes a customization language based on Xcore[7] we have used to add Trace*a* concepts as Capra extensions. Thanks to this integration you can benefit from the advanced metamodeling concepts in Trace*a* while also enjoying Capra's visualization capabilities.

[4] https://www.eclipse.org/Xtext/
[5] https://github.com/ebatot/TraceaDSL
[6] https://projects.eclipse.org/projects/modeling.capra
[7] https://wiki.eclipse.org/Xcore



```
    }

    // Available types
    relationshiptypes {
        EngineeringType Sec2PoS,
        EngineeringType NE2Class,
        DomainType Synonymy,
        DomainType Closure
    }

    // Structure of artefacts
    artefacts {
        Document doc_A { sections { sec_A_I, sec_A_II } },
        Document doc_F { sections { sec_F_I } },
        Model modelI {fragments { Cl_i  }},
        Model modelJ {fragments { Cl_j_01 , Cl_j_02 }}

    }
```

```
consoles to display at this time.
```

Fig. 4. Trace*a* Xtext plugin

In both cases, the designer can use Trace*a* as a standalone tool or add it as a new component to any model-driven pipeline, especially implemented on top of the EMF Eclipse ecosystem.

As, in complex scenarios, traces can come from different systems (using different languages or even third-party APIs), it is useful to keep Trace*a* as an *external language* that you can adapt to the changing needs of your application scenario and the types of artefacts you need to trace [11,32].

But as a trade-off, this forces designers to learn and add to their toolset a new language. An alternative option is to define Trace*a* as kind of internal DSL, embedded in a more general modeling language like SysML or UML using the extension capabilities offered by them, e.g. UML profiles.

6 Conclusion

Traceability research is scattered among different software engineering subfields resulting in diverse but partial solutions to represent traceability information. We have presented a complete traceability metamodel that aims to cover all its aspects, including the quality and uncertainty of specific traces, their decay or the evidences that support them. This information is needed to make fully informed decisions based on trace data. Our proposal has also been designed with modularity and extensibility principles in mind to facilitate its adoption in a large variety of domains. We believe it should help in improving a number of traceability-based algorithms (e.g. for impact change analysis) that could now also take into account these additional traceability dimensions.

As further work we want to continue advancing on these latter aspects, mainly proposing extensions to general modeling languages (like SysML or UML) that integrate our traceability metamodel. Moreover, we will explore the complementarity of AI and traceability. Regarding AI for traceability detection we plan to extend existing techniques to automatically infer traces to populate our metamodel considering the integrity and quality aspects of the inference process. Regarding traceability for AI, we plan to rely on our metamodel to offer better explainability support to the myriad of AI-based solutions for Software Engineering that right now mostly ignore this aspect [9,38].

References

1. Aboussoror, E.A., Ober, I., Ober, I.: Seeing errors: Model driven simulation trace visualization. In: Model Driven Engineering Languages and Systems - 15th International Conference. Lecture Notes in Computer Science, vol. 7590, pp. 480–496. Springer (2012)
2. Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E.: Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering **28**(10), 970–983 (Oct 2002)
3. Antoniol, G., Cleland-Huang, J., Hayes, J.H., Vierhauser, M.: Grand challenges of traceability: The next ten years. CoRR **abs/1710.03129** (2017)
4. Azevedo., B., Jino., M.: 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,. pp. 322–329. INSTICC, SciTePress (2019)
5. Batot, E.R., Gerard, S., Cabot, J.: A survey-driven feature model for software traceability (2020), (under review) https://github.com/ebatot/TraceaDSL/blob/main/batot2020_SurveyDrivenFeatureModel.pdf
6. Borg, M., Runeson, P., Ardö, A.: Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. Empirical Software Engineering **19**(6), 1565–1616 (2014)
7. Burgueño, L., Bertoa, M.F., Moreno, N., Vallecillo, A.: Expressing confidence in models and in model transformation elements. In: Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems. MODELS '18, ACM (2018)
8. Cabot, J., Olivé, A., Teniente, E.: Representing temporal information in UML. In: Proceedings of the 6th International Conference on «UML» - The Unified Modeling Language, Modeling Languages and Applications,. Lecture Notes in Computer Science, vol. 2863, pp. 44–59. Springer (2003)
9. Carleton, A.D., Harper, E., Menzies, T., Xie, T., Eldh, S., Lyu, M.R.: The ai effect: Working at the intersection of ai and se. IEEE Software **37**(4), 26–35 (July 2020)
10. Carniel, C.A., Pegoraro, R.A.: Metamodel for requirements traceability and impact analysis on agile methods. In: Santos, V.A.d., Pinto, G.H.L., Serra Seca Neto, A.G. (eds.) Agile Methods. pp. 105–117. Springer International Publishing (2018)
11. Cleland-Huang, J., Berenbach, B., Clark, S., Settimi, R., Romanova, E.: Best practices for automated traceability. Computer **40**(6), 27–35 (2007)
12. Cleland-Huang, J., Gotel, O.C.Z., Huffman Hayes, J., Mäder, P., Zisman, A.: Software traceability: Trends and future directions. In: Future of Software Engineering Proceedings. p. 55–69. FOSE 2014, Association for Computing Machinery (2014)

13. Dit, B., Revelle, M., Poshyvanyk, D.: Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. Empirical Software Engineering **18**(2), 277–309 (2013)
14. Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: Engineering a DSL for software traceability. In: Gasevic, D., Lämmel, R., Wyk, E.V. (eds.) Software Language Engineering. Lecture Notes in Computer Science, vol. 5452, pp. 151–167. Springer (2008)
15. Dubois, H., Peraldi-Frati, M., Lakhal, F.: A model for requirements traceability in a heterogeneous model-based design process: Application to automotive embedded systems. In: 2010 15th IEEE International Conference on Engineering of Complex Computer Systems. pp. 233–242 (March 2010)
16. Díaz, J., Pérez, J., Garbajosa, J.: A model for tracing variability from features to product-line architectures: a case study in smart grids. Requirements Engineering **20**(3), 323–343 (2015)
17. Florez, J.M.: Automated fine-grained requirements-to-code traceability link recovery. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 222–225 (May 2019)
18. Gervasi, V., Zowghi, D.: Supporting traceability through affinity mining. In: 2014 IEEE 22nd International Requirements Engineering Conference (RE). pp. 143–152 (Aug 2014)
19. Goknil, A., Kurtev, I., van den Berg, K.: A metamodeling approach for reasoning about requirements. In: Schieferdecker, I., Hartman, A. (eds.) Model Driven Architecture – Foundations and Applications. pp. 310–325. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
20. Goknil, A., Kurtev, I., van den Berg, K., Spijkerman, W.: Change impact analysis for requirements: A metamodeling approach. Information and Software Technology **56**(8) (2014)
21. Guo, J., Cheng, J., Cleland-Huang, J.: Semantically enhanced software traceability using deep learning techniques. In: Proceedings of the 39th International Conference on Software Engineering. p. 3–14. ICSE '17, IEEE Press (2017)
22. Haidrar, S., , Anwar, A., Bruel, J.M., Roudies, O.: A domain-specific language to manage requirements traceability. JSW **13**(9), 460–480 (sep 2018)
23. Heisig, P., Steghöfer, J.P., Brink, C., Sachweh, S.: 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. p. 2344–2353. Association for Computing Machinery (2019)
24. Helming, J., Koegel, M., Naughton, H., David, J., Shterev, A.: Traceability-based change awareness. In: Model Driven Engineering Languages and Systems. vol. 5795, pp. 372–376. Springer Berlin Heidelberg (10 2009)
25. Institute of Electrical and Electronics Engineers (IEEE): Ieee standard glossary of software engineering terminology. IEEE Std 610.12-1990 pp. 1–84 (Dec 1990)
26. ISO: 26262, Road vehicles – Functional safety (2011)
27. Jaber, K., Sharif, B., Liu, C.: A study on the effect of traceability links in software maintenance. IEEE Access **1**, 726–741 (2013)
28. Jiménez, Á., Vara, J.M., Bollati, V.A., Marcos, E.: Model-driven development of model transformations supporting traces generation. In: Building Sustainable Information Systems, pp. 233–245. Springer US (2013)
29. Ko, A.J., Myers, B.A.: Debugging reinvented: Asking and answering why and why not questions about program behavior. In: Proceedings of the 30th International Conference on Software Engineering. p. 301–310. ICSE '08, Association for Computing Machinery (2008)
30. Levendovszky, T., Balasubramanian, D., Smyth, K., Shi, F., Karsai, G.: A transformation instance-based approach to traceability. In: Proceedings of the 6th ECMFA Traceability Workshop. p. 55–60. ECMFA-TW '10, Association for Computing Machinery (2010)
31. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: Proceedings of the 30th International Conference on Software Engineering. p. 501–510. ICSE '08, Association for Computing Machinery, New York, NY, USA (2008)

32. Maro, S., Anjorin, A., Wohlrab, R., Steghöfer, J.P.: Traceability maintenance: Factors and guidelines. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. p. 414–425. ASE 2016, Association for Computing Machinery (2016)
33. Maté, A., Trujillo, J.: A trace metamodel proposal based on the model driven architecture framework for the traceability of user requirements in data warehouses. In: Mouratidis, H., Rolland, C. (eds.) Advanced Information Systems Engineering. pp. 123–137. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
34. Meinicke, J., Thüm, T., Schröter, R., Benduhn, F., Leich, T., Saake, G.: Feature traceability for feature-oriented programming. In: Mastering Software Variability with FeatureIDE, pp. 173–181. Springer International Publishing (2017)
35. Mikkonen, T., Nurminen, J.K., Raatikainen, M., Fronza, I., Mäkitalo, N., Männistö, T.: Is machine learning software just software: A maintainability view. In: Winkler, D., Biffl, S., Méndez, D., Wimmer, M., Bergsmann, J. (eds.) Software Quality: Future Perspectives on Software Engineering Quality - 13th International Conference. Lecture Notes in Business Information Processing, vol. 404, pp. 94–105. Springer (2021)
36. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or formal verification: Do-178c alternatives and industrial experience. IEEE Software **30**(3), 50–57 (2013)
37. Olivé, A.: Representation of generic relationship types in conceptual modeling. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Özsu, M.T. (eds.) Advanced Information Systems Engineering, 14th International Conference, CAiSE. Lecture Notes in Computer Science, vol. 2348, pp. 675–691. Springer (2002)
38. Ozkaya, I.: What is really different in engineering ai-enabled systems? IEEE Software **37**(04), 3–6 (jul 2020)
39. Pavalkis, S., Nemuraite, L., Mileviciene, E.: Towards traceability metamodel for business process modeling notation. In: Skersys, T., Butleris, R., Nemuraite, L., Suomi, R. (eds.) Building the e-World Ecosystem. pp. 177–188. Springer Berlin Heidelberg (2011)
40. Sanchez, P., Alonso, D., Rosique, F., Alvarez, B., Pastor, J.A.: Introducing safety requirements traceability support in model-driven development of robotic applications. IEEE Trans. Comput. Transactions on Computers **60**(8), 1059–1071 (2011)
41. Sannier, N., Baudry, B.: Toward multilevel textual requirements traceability using model-driven engineering and information retrieval. In: 2012 Second IEEE International Workshop on Model-Driven Requirements Engineering (MoDRE). pp. 29–38 (Sep 2012)
42. Seibel, A., Hebig, R., Giese, H.: Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance, pp. 215–240. Springer (2012)
43. Spanoudakis, G., Zisman, A., Pérez-Miñana, E., Krause, P.: Rule-based generation of requirements traceability relations. Journal of Systems and Software **72**(2), 105 – 127 (2004)
44. Taromirad, M., Paige, R.F.: Agile requirements traceability using domain-specific modelling languages. In: Proceedings of the 2012 Extreme Modeling Workshop. p. 45–50. XM '12, Association for Computing Machinery (2012)
45. Wang, F., Yang, Z., Huang, Z., Liu, C., Zhou, Y., Bodeveix, J., Filali, M.: An approach to generate the traceability between restricted natural language requirements and aadl models. IEEE Transactions on Reliability **69**(1), 154–173 (2020)
46. Winkler, S., von Pilgrim, J.: A survey of traceability in requirements engineering and model-driven development. Software and Systems Modeling **9**(4), 529–565 (2010)
47. Wohlrab, R., Knauss, E., Steghöfer, J.P., Maro, S., Anjorin, A., Pelliccione, P.: Collaborative traceability management: a multiple case study from the perspectives of organization, process, and culture. Requirements Engineering **25**(1), 21–45 (2020)
48. Yrjönen, A., Merilinna, J.: Tooling for the full traceability of non-functional requirements within model-driven development. In: Proceedings of the 6th ECMFA Traceability Workshop. p. 15–22. ECMFA-TW '10, Association for Computing Machinery (2010)