2nd Deliverable: Traceability language – definition and editor

Edouard R. Batot

SOM Research Lab. Office 223
Av. Carl Friedrich Gauss, 5. Building B3
08860 Castelldefels (Barcelona), Spain

CONTENTS

LIST OF ACRONYMS

MDE	Model-Driven Engineering
RE	Requirement Engineering
UML	Unified Modeling Language
DSML	Domain Specific Modelling Language
XMI	XML Metadata Interchange
AI	Artificial Intelligence
ML	Machine Learning
EA	Evolutionary Algorithm
NLP	Natural Language Processing
PoS	Part-of-Speech
SOM	Systems, Software and Models Lab
UOC	Open University of Catalonia
CEA	Commissariat à l’énergie atomique et aux énergie alternatives

FOREWORD

"La connaissance, dans une organisation quelle qu'elle soit, notamment une entreprise, n'est pas visible en tant que telle. Elle n'est visible qu'à travers certaines traces (des informations, des documents, des discussions, des groupes de personnes etc.). Ces traces se rapportent toujours à un élément de l'entreprise (une machine, un processus, une expérience, une unité de production etc.). Elles représentent cet élément en tant que système actif et relativement stable de l'entreprise, système complexe dans le sens où il est difficile d'en avoir une vision intelligible, tant ces traces peuvent être nombreuses, dispersées, sans cohérence apparente. C'est ainsi que se forme la connaissance, à partir de la perception de ces systèmes qui crée ses traces, pour mieux les comprendre et mieux les maîtriser."

Une théorie structurelle de la connaissance,
Jean-Louis Ermine

1 INTRODUCTION

The purpose of this report is to document the work done as part of the Second Task described in the Convention de l'équipe de recherche commune. This document corresponds to Deliverable 2. The core contribution of this document is a domain-specific language (DSL) for traceability, expressive enough to cover all the potential traceability applications we have discussed during this project.

More specifically, we include a short reminder of the terminology we use in the DSL, the definition of both the abstract (*i.e.,* metamodel) and concrete syntax (a JSON-based representation), and an Ecore implementation of the DSL in an Eclipse plugin. We illustrate the usability of the DSL with a running example: the transclusion of model elements in certification documents.

2 TERMINOLOGY

In this section, we review a set of term definitions that will help to grasp the detail of our DSL conceptualization. These definitions aim to encompass and unify all the different uses and visions of traceability. They do not all appear directly in the metamodel, nor do they aim to cover all the different visions on traceability, but they provide a common generic core that shall be then adapted to specific scenarios or specific use cases for the DSL.

– **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 [10] as

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

– A **trace** is a path from one artefact to another. Composed of atomic **links** that directly relate artefacts with each others. The heterogeneous nature of traceability applications confers relationship types a peculiar attention since it helps understand the rationale behind relationships - it informs not only *how* artefacts are linked but also *why* [12]. Typing is a primary concern in conceptual modeling in general [13]. Here, the **type of a relationship** informs about its semantics in its application domain.

– A **traceability information model**(TIM), or traceability metamodel, defines the representation of traces, their data structure and behaviour, [5]. In both cases it defines at the language level the concepts and relationships available for tracing. With the manifolds of traceability purposes, no common representation has emerged yet which is one of the key motivations for this work.

– A **referee** is the (human) actor responsible for a traceability artefact.

– A **link** is a concrete relationship between two artefacts.
  Note that most of the work aiming at modeling "traceability" actually models traceability links. This suits the OMG standard definition of traceability which consider the only input/output of transformations [20]. In UML or SysML, a *link* is a specialization of the concept of Dependence (which is itself a specialization of DirectedRelationship) which is used to explicitly model a traceability relation between two sets of elements.

– **Explicit links** refer to artefacts that explicitly link to each other in the concrete syntax.

– **Implicit links** show artefacts bondage at a syntactic or semantic level without explicit link (*e.g.,* binary class and their respective source code artefact are implicitly "linked" to each others) [14]. For pragmatic purposes, these links can become *materialized*.

- An **artefact** can be any element of a system - e.g. unstructured documentation, code, design diagrams, test cases and suites... The nature of artefacts follows two main dimensions: the lifecycle phase they belong to (e.g. specification, design, implementation, test), and their type (e.g. unstructured natural language, grammar-based code, model-based artefact). The **granularity of artefacts** is the level to which artefacts can be decomposed into sub parts. We call a **fragment**, the resulting product of the decomposition of an artefact. A fragment can be itself broken down into smaller parts (or sub-fragments), and so forth. For example, a software requirement document could include a guideline for certification which in turn could include sub-sections and a model-level definition of their implications.
- **Trace integrity** is the degree of reliability that bares a trace. It is an indirect measure that includes, for example, both the age of a trace, the volatility of artefacts targeted by the trace, and the automation level of tracing features. This indication is supported by **evidences** that can be quantitative or qualitative. For example, how long (how many versions ago) has the trace been identified in the system? Or, has the trace been identified manually or automatically? Is there an automated co-evolution mechanism between traces and targeted artefacts? What is the level of experience of the trustee who identified it? The volatility of source and target artefacts are also factors that may influence the relevance and accuracy of a trace.
- **Pre-requirement and post-requirement** traceability refer to, respectively, traces identified during specifications elicitation and during the implementation (design and code) step of a specification [8]. The IEEE Guide for Software Requirements Specifications mentions **forward** and **backward** traceability, referring to the ability to follow traceability links from a source to a specific artefact, or the opposite from the artefact to its source respectively [9] but, technically, the direction of traceability link (from source to target, or from target to source) does not make a difference.
- **Vertical traceability** refers to the linkage between artefacts at different levels of abstraction (*e.g.,* derives, implements, inherits) whereas **horizontal traceability** refers to artefacts at the same level (*e.g.,* uses, depends on).
- **Time related traceability** goes along two dimensions: the evolution of (a group of) elements through successive development tasks, or the evolution of artefact properties during an execution of the system.

3 TRACEA – A DSL FOR TRACEABILITY

A DSL[1] is defined through three main components [11]: abstract syntax, concrete syntax, and semantics.

The abstract syntax defines both the language concepts and their relationships, and also includes well-formedness rules constraining the models that can be created. Meta-modeling techniques are normally used to define the abstract syntax. The concrete syntax defines a notation (textual, graphical or hybrid) for the concepts in the abstract syntax, and a translational approach is normally used to provide semantics, though most of the time it is not explicitly formalized.

The metamodel definition directly includes some of the terminology discussed in the previous section. For other concepts we have made sure the DSL is complete enough to allow for that kind of processing or analysis. For instance, the concept of traces transcribes directly into classes of the metamodel (*e;g.,* traces, links, artefacts). Processes span over subsets of classes (*e.g.,* granularity and typing). Characteristics such as vertical traceability, or implicit traces do not appear directly in the metamodel but are either found disseminated among its structural composition or the DSL elements allow for their implementation in specific use cases.

In this section,

- We provide a running example: transclusion for traceability between certification documents and design models.
- We present the metamodel, decomposed into five sub-areas of traceability and detail each of them. We reuse the example to show the soundness of our choices.
- We provide a textual syntax in the form of a JSon-based grammar.

3.1 ILLUSTRATIVE EXAMPLE: CERTIFICATION AND DESIGN TRANSCLUSION

Certification documents are external to the software system they are certifying. This means that when the software evolves, the content of these documents must be reevaluated to see if the new version of the system can still satisfies the certification conditions. Ideally we would only like to examine the certificate against the changes. And similarly for any change in the certification rules or conditions. We would like that only those parts of the system potentially being affected need to be reviewed and adapted if necessary. To be able to support this "incremental" evaluations and facilitate the co-evolution of certification

[1]We use the DSL term to refer to Domain-Specific Modeling Languages, which is the subset of DSLs that are relevant to us

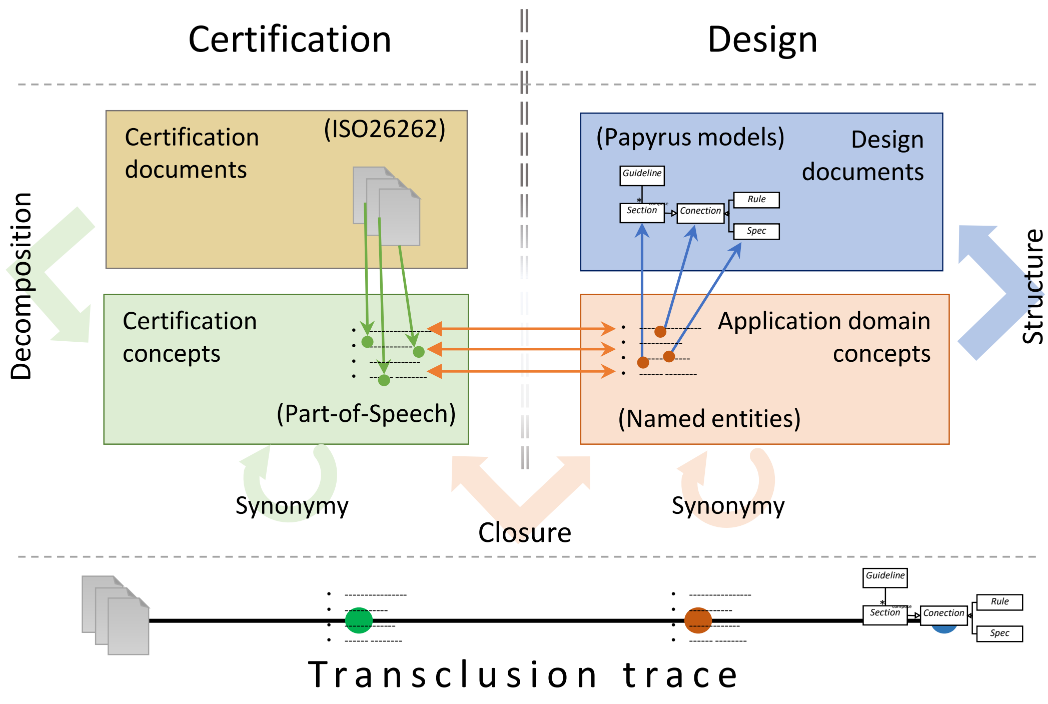


FIGURE 1: TRANSCLUSION OF PART-OF-SPEECH TO MODEL ENTITIES.

and design artefacts, we need to keep track of the links between the nominal expressions in certifications documents and the elements of design models they target or impact.

As can be seen on the certification side in Figure 1, nominal expressions form semantically rich patterns (*e.g.,* nominal groups, action verbs, nouns, acronyms). They are spread among the sections of the certification documents. The patterns of Parts-of-Speech (PoS) can be extracted using natural language processing techniques[2]

On the design side, named entities are model entities characterized by a unique name (*e.g.,* a method, a class, a package, a model). The structural hierarchy represented by Papyrus models can be used to cluster and disambiguate its constitutive entities [15]. Named entities have synonyms that must be considered as well - and disambiguated when possible. The association between PoSs and model entities is established with a neural model capable of measuring the (semantic) distance between the elements of both realms[3]. Certification documents are semi-structured text artefacts. Their implementation and validation scenarios may correlate in many different ways with their corresponding modelling

[2]The quality of PoSs vary depending on the technique used, its parameterization, as well as the dataset used for training. These choices remain mostly arbitrary and evidences of the superiority of one algorithm or method upon the others remain haphazard [21]. In this report, we aim at conceptualizing the problem domain, we do not provide details on the choice and parameterization of such techniques.

[3]An interesting methodology to do so is to train a ”general intelligence”, or to ”generally train” a model from available vast corpuses in (more or less) natural language. Then this model is refined with corpuses relevant to the application domain the system belongs to [17].

entities. We chose the most straight forward case to illustrate an application of the Tracea metamodel: the transclusion of `PoS` to `NamedEntities`. Figure 1 shows a high level representation of a this process.

This example helps grasp the details of the application of our traceability language to a specific domain: an explicit record of the link between important elements of certification and the entities they target in Papyrus models to allow direct transclusion[4].

Transclusion traces use vertical links (from documents to PoS, and from model to named entity), and horizontal links (between the synonyms and the closure between PoSs and named entities). Links related to the composition are explicit: they are contained in the syntax of the domain (*e.g.,* classes are *part* of a package); whereas synonyms and closure are implicit (*i.e.,* they do not appear in the syntax). This example has a post-requirement nature as it is established after the requirements have been determined.

3.2 METAMODEL

3.2.1 CORE STRUCTURE

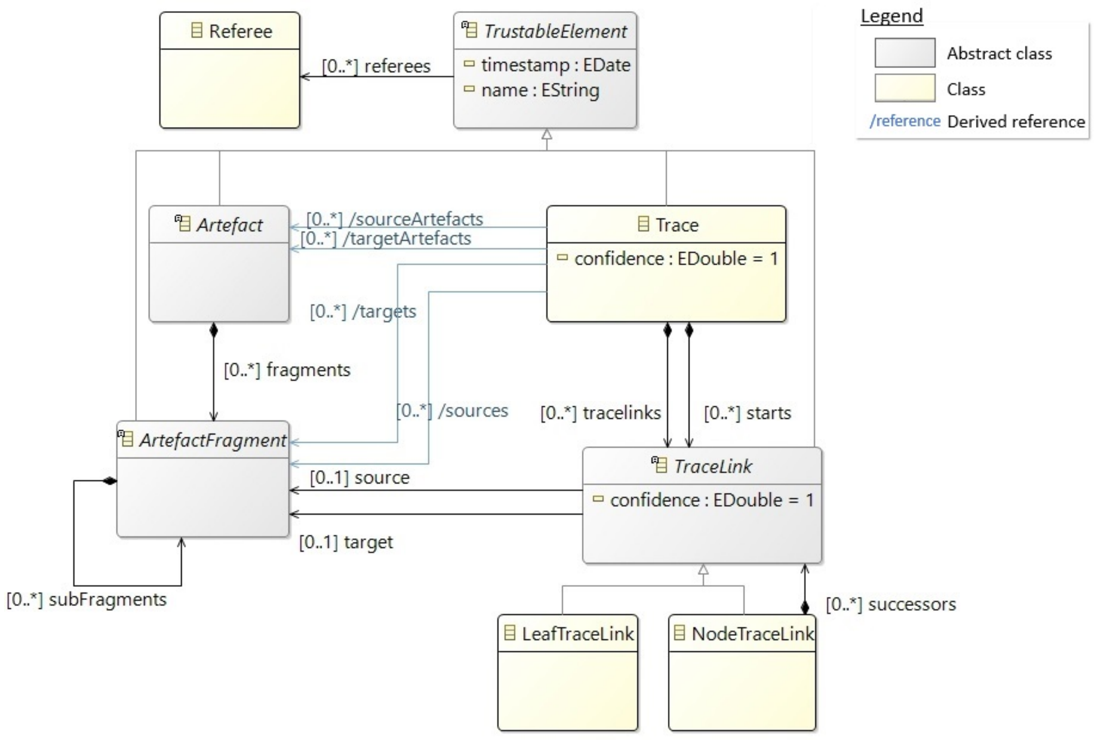


FIGURE 2: CORE STRUCTURE OF TRACEA METAMODEL

[4]A transclusion is the inclusion of part or all of a document into one or more other documents by hypertext reference.

The core of traceability is to arbitrarily bind artefacts of a system(s) with each others. A trace may have one or many sources and ends with one or many targets. The connection among the artefacts in a trace is expressed as the combination of atomic trace links representing direct connections between a number of fragments of each involved artefact.

Fig. 2 shows an excerpt of Tracea metamodel. This excerpt describes the core structure of traces. It describes the composition scheme of its links (bottom right corner), offers an interface for the decomposition of artefacts into fragments (left side), and allows to record information related to the authorities potentially accountable for its constituents (top class).

3.2.1.1 CLASSES

- A `Trace` is a composition of `TraceLinks`. It references the source of trace links (*starts* reference) as well as the trace links involved to access the different artefacts (*tracelinks* reference). A trace also references to source and target artefacts and fragments. These references are derived from the `TraceLinks`.
- A `TraceLink` is the fundamental structure to build the trace sequences and branches. It is a direct connection between two elements of the system (*i.e.,* `ArtefactFragments`). A `TraceLink` has one source fragment and one target fragment. It is also the abstract node of a tree of `TraceLinkNode` and A `TraceLinkLeaf` refers to a list of successors being part of the linkage of the trace. This class is abstract and is refined into `NodeTraceLink` and `LeafTraceLink` following the Composite design pattern.
- A `NodeTraceLink` is a node in a tree like `TraceLink`. It references successor links.
- A `LeafTraceLink` is a leaf in a tree-like `TraceLink`. It has no successor link reference.
- An `Artefact` is an abstract representation of an element of the traced system. An artefact represent a text document, a class diagram, or any other kind of document. (See Section 3.2.2)
- A `ArtefactFragment` is a part of an Artefact. It can be further decomposed into sub fragments for finer granularity. (See Section 3.2.2)
- A `TrustableElement`: Is an abstract class representing elements that have am identifier (name) and a timestamp for timed traceability consideration. It also references `Referees` to its specializations (*i.e.,* `Trace`, `TraceLink`, `Artefact`, `ArtefactFragment`, `Evidence` - see Section 3.2.4 for details).
- A `Referee` is an actor accountable for a `TrusteableElement` (See Section 3.2.4)

3.2.1.2 WELL-FORMEDNESS RULES

Well-formedness rules are expressed as OCL constraints. Rules in Figure 3 ensure that the inclusion of starters in the list of links a trace have, and the derivation of derived references related to source and target artefacts and fragments.

```
context Trace inv starters:
        self.tracelinks –>includesAll(self.starts)

context Trace inv derivedSources:
        self.tracelinks–>collect(source)
        –>includesAll(self.sources)

context Trace inv derivedTargets:
        self.tracelinks –>collect(target)
        –>includesAll(self.targets)

context Trace inv derivedSourceArtefacts:
        self.tracelinks –>collect(source)
        –>includesAll(self.sourceArtefacts.fragments)

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

FIGURE 3: OCL CONSTRAINTS OVER ELEMENTS OF THE CORE PACKAGE.

3.2.1.3 ILLUSTRATIVE EXAMPLE

The core section of the metamodel aims at representing the structure of the linkage between the artefacts that are part of a trace. In our example, a trace sequentially links the PoSs in certification documents to their corresponding entities in design models. In simple words, a document contains Parts-of-Speech (PoSs) where each PoS is associated to a number of model elements, themselves contained in one or more models. These directions are `TraceLinks`. Figure 4 shows a diagrammatic perspective of the structure of a transclusion. The example transclusion trace relates the PoS (b, c, and d) of documents (a and f) to the `NamedElements` (i and j) of three Papyrus models (e, g, h). We see on the right side that some `TraceLinks` have successors, they are `TraceLinkNodes`; when they do not they are `TraceLinkLeafs`.

When represented as instances of the Tracea metamodel, a certification document is

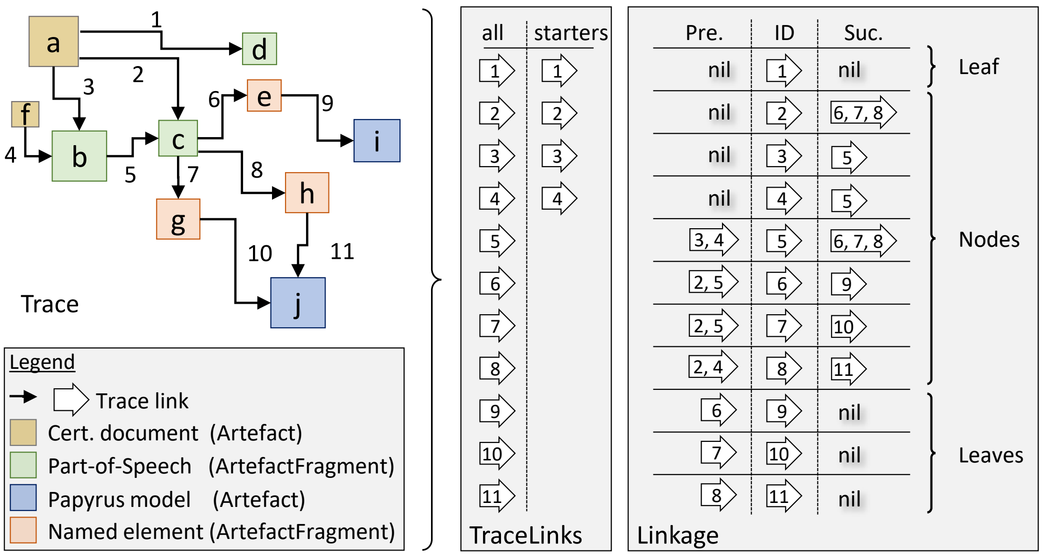


FIGURE 4: STRUCTURE OF A TRANSCLUSION TRACE.

an Artefact, a section of a document is an `ArtefactFragment`. A PoS is a fragment as well. On the model side, models are `Artefacts`. Packages, classes, and named elements are `ArtefactFragments`. We will develop on relationship types in Section 3.2.3.

3.2.2 ARTEFACT GRANULARITY

Project stakeholders must decide on the appropriate level of trace granularity for each artifact type. For example, when tracing to UML class diagrams, they could generate a trace at the package, class, or method level. Granularity must be carefully determined to effectively support stakeholders in their traceability tasks, while minimizing the effort involved to analyze and utilize the set of returned trace links. This can be especially problematic in large, weakly structured documents that might not contain clearly defined components at the desired granularity level. To mitigate this problem, automated traceability tools can cluster sentences into meaningful semantically related groups and then generate traces to those groups [4].

As can be seen in Figure 5, We distinguish between five main types of artefacts: i) natural language and unstructured text documents are `TextArtefacts`, ii) executable test unit and suites are `TestArtefacts`, iii) models at design or conceptual level are `ModelArtefacts`, and iv) source code document, SQL sources, scripts and configuration documents are `CodeArtefacts`. They are all specialization of Artefact. Types differ in the nature of the structure of the artefacts. This is an arbitrary separation grounded on our experience with traceability. Other can be easily introduced to span closer to user needs. An `ArtefactFragment` is a part of an Artefact. It can be subdivided into sub-fragments.

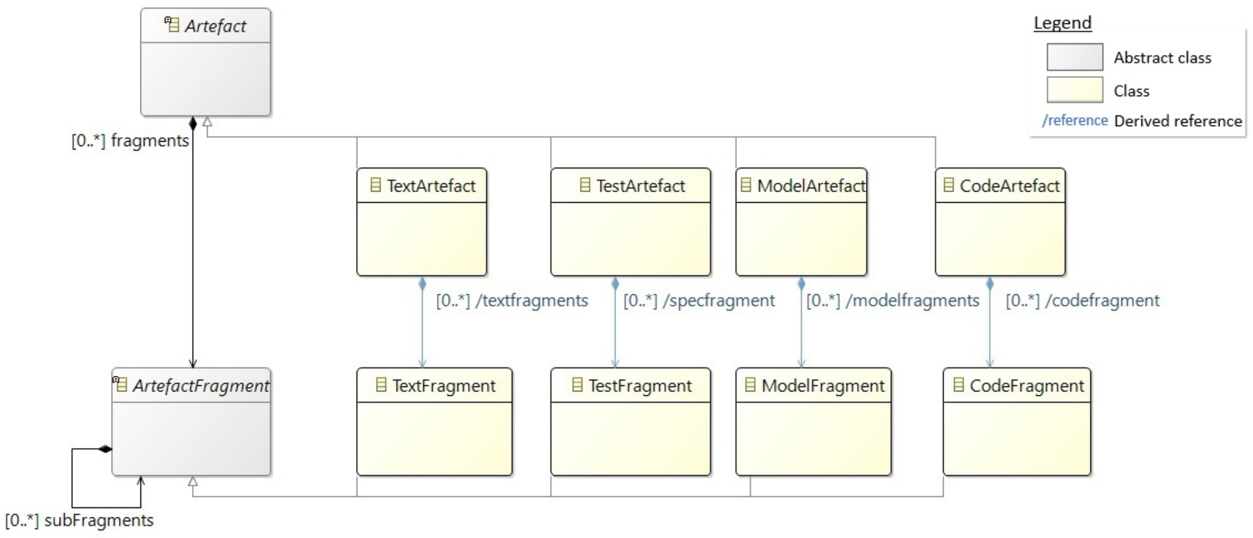


FIGURE 5: CUSTOMIZABLE GRANULARITY

Note that we could also opt for a rather simpler approach where artefact types are not modeled as subclasses but where we would just have a single attribute `ArtefactType` in `Artefact`. The obvious disadvantage with this option is that we lose the opportunity to add information relevant only to specific types of artefacts.

3.2.2.1 CLASSES

- A `TextArtefact` is a(n) (un)structured text document. `TextArtefacts` can further decompose into *Sections*, *Subsections*, and *Paragraph*. Each of them may contain *part-Of-Speech* that points to concepts in the application domain.
- A `ModelArtefact` is an artefacts related to a model-level representation. `ModelArtefacts` can be UML, SysML or other types of models, with a structural hierarchy inherent to their entity-relation form. In the case of UML, a decomposition may follow the *Package*, *Class*, *Methods*, *Attributes*, and *References* hierarchical scheme. *Named Elements* point to unique elements of the language uniquely identified.
- A `CodeArtefact` is a documents containing source code independently of the specific programming language. `CodeArtefacts` may decompose into *CodeBlock*, *Header*, *Variable*, *Methods*, and so on.
- A `TestArtefact` is a test unit or suite.
- And so forth and so on for their respective fragments.

3.2.2.2 WELL-FORMEDNESS RULES

Well-formedness rules are expressed as OCL constraints. Rules in Figure 6 the coherence between subtypes of artefacts, fragments, and subfragments.

```
context TextArtefact inv textFragmentsTyping:
    self.fragments −>forAll(self−>oclIsKindOf(TextFragment)

context CodeArtefact inv codeFragmentsTyping:
    self.fragments −>forAll(self−>oclIsKindOf(CodeFragment)

context ModelArtefact inv modelFragmentsTyping:
    self.fragments −>forAll(self−>oclIsKindOf(ModelFragment)

context TestArtefact inv testFragmentsTyping:
    self.fragments −>forAll(self−>oclIsKindOf(TestFragment)

context TextFragment inv textsubfragmentsTyping:
    self.subfragments −>forAll(self−>oclIsKindOf(TextFragment)

context CodeFragment inv codesubfragmentsTyping:
    self.subfragments −>forAll(self−>oclIsKindOf(CodeFragment)

context ModelFragment inv modelsubfragmentsTyping:
    self.subfragments −>forAll(self−>oclIsKindOf(ModelFragment)

context TestFragment inv testsubfragmentsTyping:
    self.subfragments −>forAll(self−>oclIsKindOf(TestFragment)

context NamedElement inv noNestedNamedElements:
    self.namedelementsDefined −>isEmpty() and
    self.namedelementsUsed −>isEmpty() and
    self.subfragments −>isEmpty()
```

FIGURE 6: OCL CONSTRAINTS OVER ELEMENTS OF THE GRANULARITY PACKAGE.

3.2.2.3 ILLUSTRATIVE EXAMPLE

A trace starts with the PoSs we want to trace down to the design level. It also contains the successive containment connections from document, to section, to PoS in order to refine the granularity of the results to sections instead of whole documents. The trace also contains the link between PoSs and their corresponding model entities, as well as model entities hierarchy (*e.g.,* the class, method, or package they belong to).

To summarize:

- A text `Document` is a `TextArtefact`. It is decomposed into `Sections`. Sections define or use `PoSs`. Both Sections and PoSs are `TextFragment` (derived from `ArtefactFragment`).
- A Papyrus `Model` is a `ModelArtefact`. It is decomposed into modelling elements : *e.g.*, `Packages`, `Classes`, and `Structural features`. They are `ModelFragments`. `ModelFragments` (spceializations of `ArtefactFragments`) define and use `NamedElements` (*i.e.*, elements of the system with unique identifier).

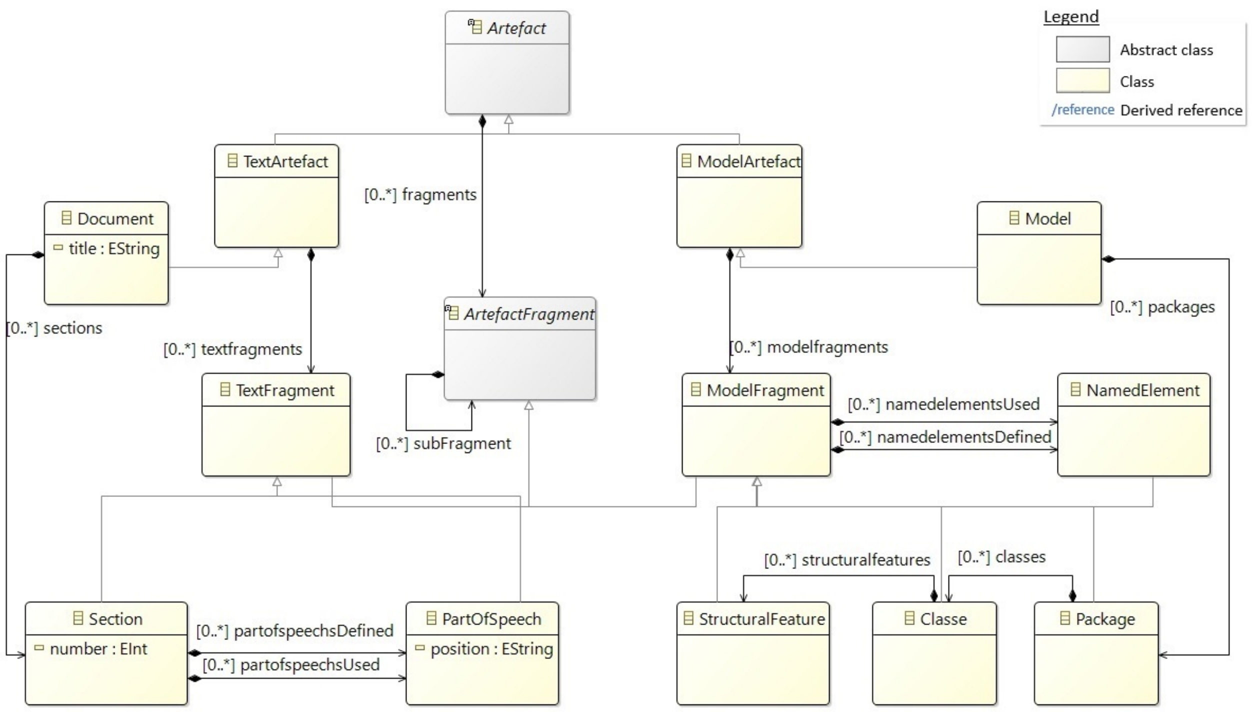


FIGURE 7: GRANULARITY CUSTOMIZED FOR CERTIFICATION TO MODEL TRANSCLUSION

3.2.3 RELATIONSHIP TYPING

Stakeholders with different perspectives, goals and interests who are involved in software development may contribute to the capture and use of traceability information. Depending on their expertise and needs, they may come up with different types of traceability relations expressing different perspectives on the links among the elements in the project. It is very important to understand project specific conventions for interpreting the meaning of such relations [18] as their semantics will make any traceability analysis much richer and precise.

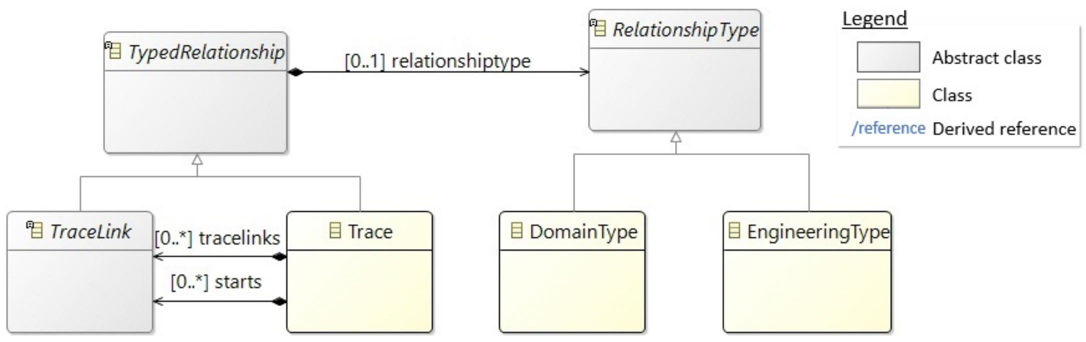


FIGURE 8: CUSTOMIZABLE RELATIONSHIP TYPES

As can be seen in Figure 8, we refine the typing of relationships with a mechanism comparable to UML data typing [16]. A `TypedRelationship` has an optional `RelationshipType`. A `RelationshipType` is abstract and specializes into different levels of sub types. At the top level, `DomainType` and `EngineeringType` define predefined `RelationshipType`, without any substructure.

On the one hand we have `DomainType` types coming from the application domain, used by experts of this domain. They convey semantics of the application domain. On the other hand, `EngineeringType` are types related to the engineering of software systems and focus on "how" artefacts are related (*i.e.,* implements, Doc2Vec). We aim to reflect in this architecture the cognitive gap that remains between traceability actors and the users of their product.

This is an arbitrary separation and other specific types can be easily added to this general design, same as we had for the artefact types. As any metamodel, this one can also be extended to better suit the needs of a specific domain. Some of RelationshipTypes could be generic enough to be reused in different scenarios while others will be very scenario-specific.

3.2.3.1 CLASSES

- A `TypedRelationShip` is an abstract class for the representation of relationships with a type (optional).
- A `RelationshipType` is an abstract class representing a type of relationship.
- A `DomainType` is a `RelationshipType` dedicated to application domains (*e.g.,* Transclusion).
- An `EngineeringType` is a `RelationshipType` dedicated to engineering domains (*e.g.,* implements, derives, uses).

3.2.3.2 CONSTRAINTS

No constraints.

3.2.3.3 ILLUSTRATIVE EXAMPLE

The types of the links are visible in Figure 9. They are refinement of `RelationshipType`. The three specialized types are `Transclusion`, `Closure`, and `Synonymy`. These are `DomainType` with semantics related to the reason why two artefacts relate with each other. A `Transclusion` is a complex `DomainType`. It is a `Trace` composed of links of different types. We see in Section 3.2.4 that `Evidences` are used to record and store parameters and configurations of these more complex kinds of links.

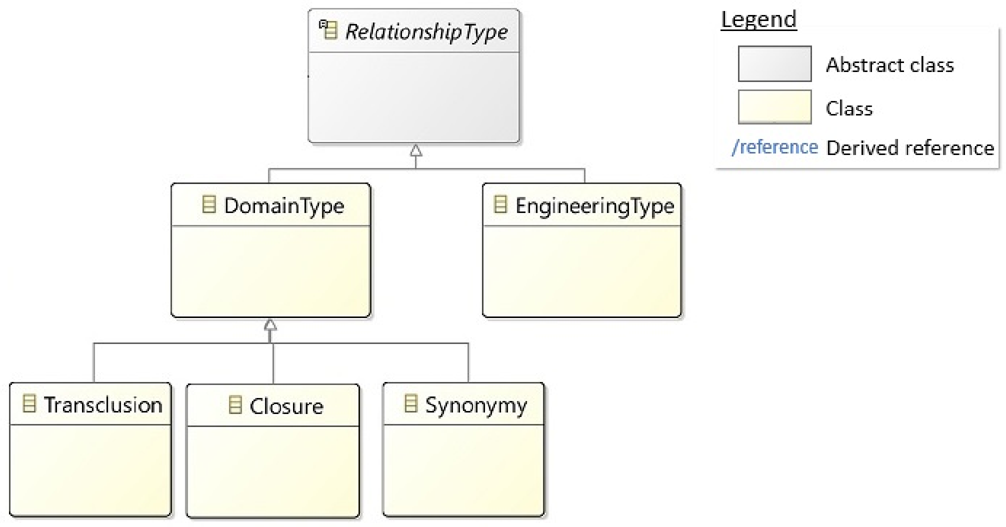


FIGURE 9: CUSTOMIZED RELATIONSHIP TYPING TAILORED TO TRANSCLUSION.

3.2.4 INTEGRITY & ACCOUNTABILITY

When looking at traces is critical to be able to answer questions like ”Who is responsible for the creation of this trace?”; ”To whom should I refer for more information?”; ”How much can I trust this trace is correct?”,... If a clear answer is not available, we may lose faith in the whole traceability system or make wrong assumptions on the traceability data that could have very negative impact on the future evolution analysis of the system.

This is why we believe an explainability mechanism should be a first-class citizen in any traceability approach [7]. Therefore, Tracea brings a dedicated approach to collect, structure, and handle this information.

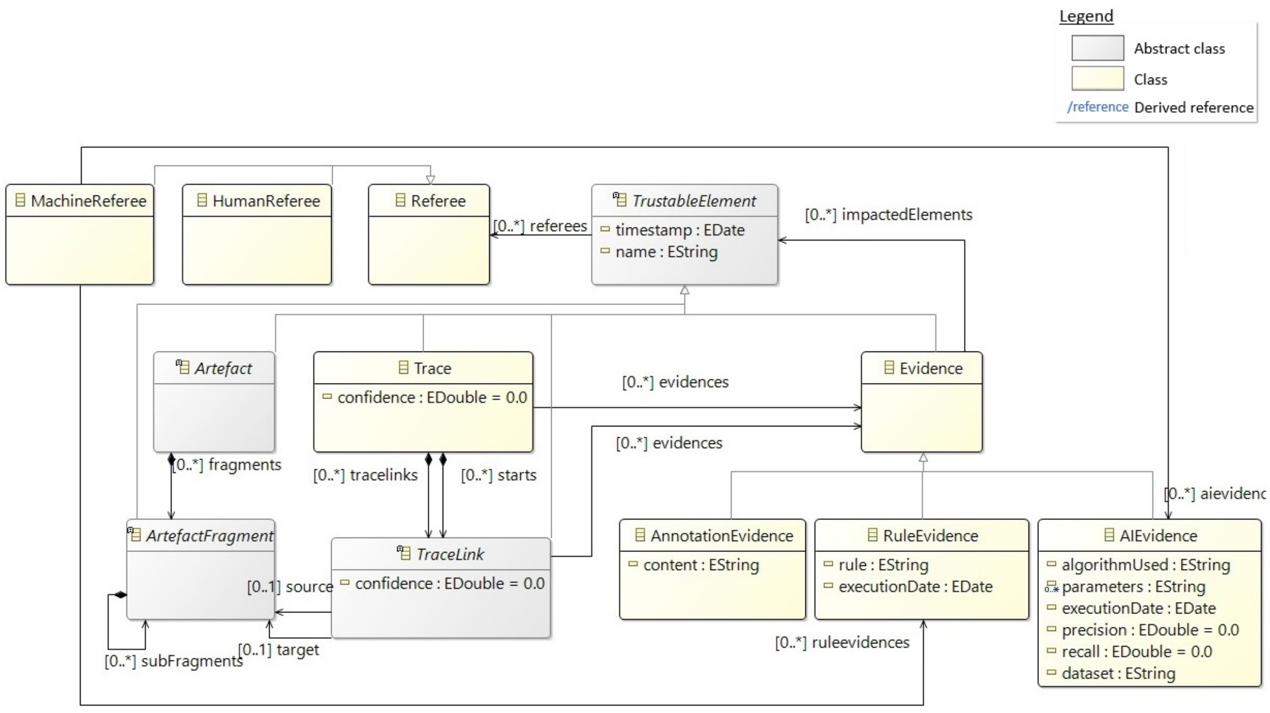


FIGURE 10: INTEGRITY

To begin with, every elements of the Tracea metamodel extends a `TrusteableElement` (see Figure 10) as we have seen before. A `TrusteableElement` contains references to `Referees` accountable for this element, together with a unique name, and a timestamp.

Beyond this basic support, we also collect evidences for each trace that explain how (and based on what information) that specific trace was created. As such, the `Evidence` metaclass enables expressing the sets of facts that testify on the means of elicitation, creation, or modification of each elements of a trace. A `Trace` is potentially linked to a set of evidences. `Evidences` are specialized in three subclasses. A trace could be generated as a result of the analysis of a text annotations with unstructured or grammar-based text explaining the rationale behind such trace (`AnnotationEvidence`). Or it could be the result of executing a rule (`RuleEvidence`). Finally, traces can also be the result of the execution of machine learning algorithms and store the type, the ID of the training set, and the precision and recall obtained on that set (`AIEvidence`).

`Evidences` increase our certitude that the trace actually exists. Yet, some uncertainty may remind that reflects our belief. We model the (un)certainty a user may have on the quality/existence of the traces with a *confidence* attribute for `Traces` and `TraceLinks`. It is a first representation of all the levels of uncertainty that can affect traces (more details in [3]).

3.2.4.1 CLASSES

- An **Evidence** is an element that participate in the evaluation of trace reliability. An Evidence references a set of impacted TrusteableElements.
- An AIEvidence is an Evidence based on the parameterization and learning precision and recall of an AI algorithm. For example, with traces automatically identified using a machine learning algorithm, an evidence contains the type of the algorithm, its parameters and the level of confidence (*i.e.,* precision and recall on a test bench).
- A RuleEvidence is an Evidence describing the execution of integrity rules and their results with a bond to the elements they impact.
- An **AnnotationEvidence**: Textual annotation accounting for Evidence.
- A Referee is an actor accountable for a TrusteableElement.
- A MachineReferee is a non-human actor responsible for a TrusteableElement (*i.e.,* when generated or derived automatically).
- A HumanReferee is a human actor responsible for a TrusteableElement.

3.2.4.2 CONSTRAINTS

No constraints.

3.2.4.3 ILLUSTRATIVE EXAMPLE

In our example, PoSs are extracted from certification TextDocuments using natural language processing (NLP) techniques. A PoS2NamedEntity links a PoS and a NamedEntity based on the evaluation of their semantics using machine learning techniques as well. These methods need to be adapted and parameterized, and concrete datasets must be used to train the learning models. We store these attributes in instances of AIEvidence classes. These records will help understand the evolution of the automation. As can be seen in Figure 11, we instantiate AnnotationEvidence to record the nature of the synonymy between PoSs and between NamedEntities. A PoSSynonym (resp. NamedEntitySynonym) relates two (or more) PoS (reps. NamedEntity) and record the provenance of the evaluation of the distance between the set of elements at play.

Referees are created for each Artifact and Trace instantiated or modified. The name and the timestamp will provide the useful information to retrieve who is responsible for these elements.

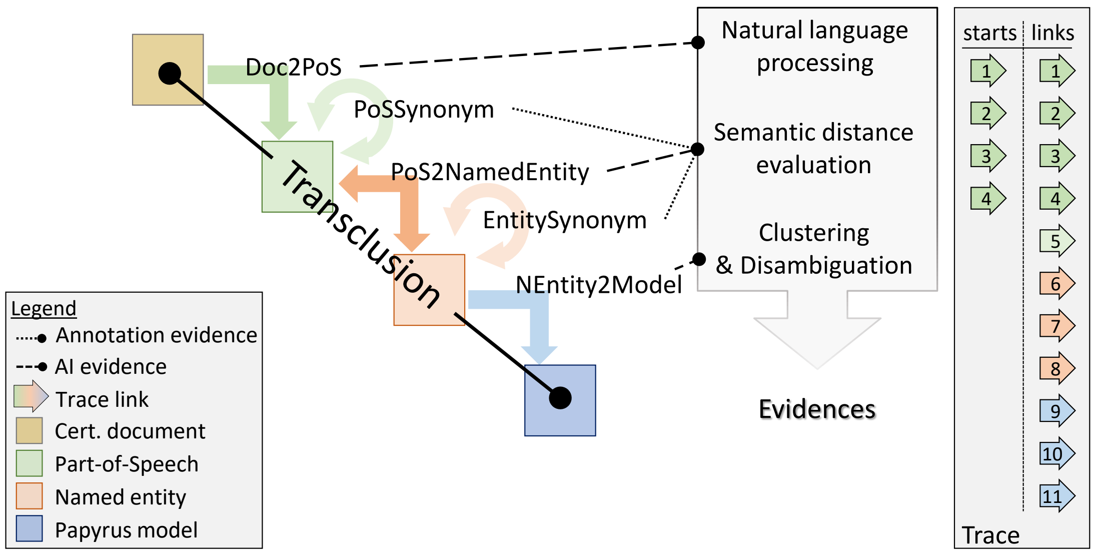


FIGURE 11: COMPLETE TRANSCLUSION PROCESS AND THE RECORD OF EVIDENCES.

3.3 TEXTUAL SYNTAX

We chose to use a JSon-like grammar as a first generic textual syntax of Tracea. We believe this type of concrete syntax is appropriate. It is verbose, but *i)* it is readable by humans without holding any structural information from the abstract syntax, and *ii)* known generators can be easily adapted to generate instances. The complete grammar definition has been uploaded on the Modelia Git repository [1].

As can be seen in the excerpt parts of Figure 12, 13, and 14, classes and structural features of the metamodel are keywords of the grammar. Qualified named are used to navigate into inner elements (*i.e.*, following the pattern `'Root.element.innerElement'` structure with the dot as connector). Elements of a list are set between brackets `{}` and separated with a coma `,`.

In Figure 12 `Traces` are declared sequentially. They contain the declaration of their `TraceLinks`. Figure 13 shows examples of `RelationshipTypes` declarations, and `Artefacts` and `Fragments` decomposition. These elements can be later referenced in different elements of the system. Their dependence level to other elements is weak. In the same manner, Figure 14 presents two `Evidences` declarations and two `Referees`.

3.4 CONCLUSION AND FUTURE WORK

In this deliverable, we have developed a DSML for traceability. In the future, this DSL could be extended with the consideration of belief uncertainty as addressed by Loli Burgueno in the Modelia project [3]. Traceability has an important human factor to consider. This exten-

```
// Structure of traces (Excerpt)
    traces {
        Trace Transclusion1 {
            tracelinks {
                LeafTraceLink link01 {
                     source sec_A_I
                     target PoS_D
                     relationshiptype Sec2PoS
                     evidences {Evidence_link01}
                     referees (Sebastien)
            },
                NodeTraceLink link02 {
                     source sec_A_II
                     target PoS_C
                     successors {link06,link07,link08}
                     relationshiptype Sec2PoS
                     evidences {Evidence_link02}
                     referees (Rd15OUA5RD)
            },
// ...
        }
    }
```

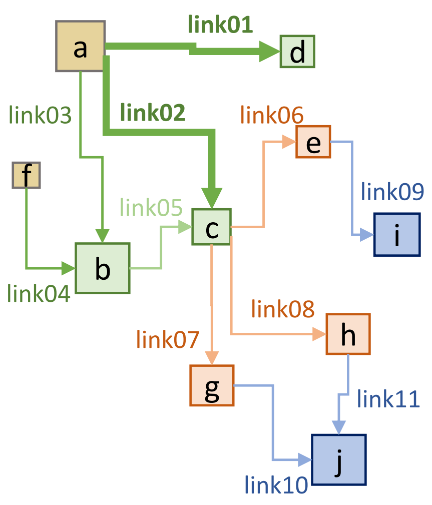


FIGURE 12: TEXTUAL SYNTAX EXCERPT: CORE STRUCTURE OF A TRACE.

sion would allow reasoning, manipulation and decision making with belief and subjective opinions of the referees involved in the process.

Then, since time plays a fundamental role in the decline of coherence in a system, this dimension shall be investigated meticulously as well. Coordinating the record of tracing operation time wise would give the opportunity to evaluate the aging of traces. In this first version, we simply record *timestamps* for all trusteable artefacts. These stamps shall be augmented with *TemporalEMF*, a profile for temporal metamodelling developed in SOM specifically for EMF solutions [6].

Finally, another interesting follow up would be to adapt this DSL to UML, SysML or DOORS in order to populate it with real world traces. This future work requires the evaluation of entry points from Tracea to other languages. It also requires a definition of the purpose of tracing in the projects targeted.

```
// Relationship 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 }}
}
```

```
// Structure of fragments
fragments {
    Section sec_A_I { partofspeechsDefined {PoS_D }},
    Section sec_A_II { partofspeechsDefined {PoS_B, PoS_C}},
    Section sec_F { partofspeechsDefined {PoS_D, PoS_B, PoS_C}},
    PartOfSpeech PoS_B { },
    PartOfSpeech PoS_C { },
    PartOfSpeech PoS_D { },
```

```
    Classe Cl_i { namedelementsDefined {ne_E} },
    Classe Cl_j_01 { namedelementsDefined {ne_G} },
    Classe Cl_j_02 { namedelementsDefined {ne_H}},
    NamedElement ne_E { },
    NamedElement ne_G { },
    NamedElement ne_H { }
}
```

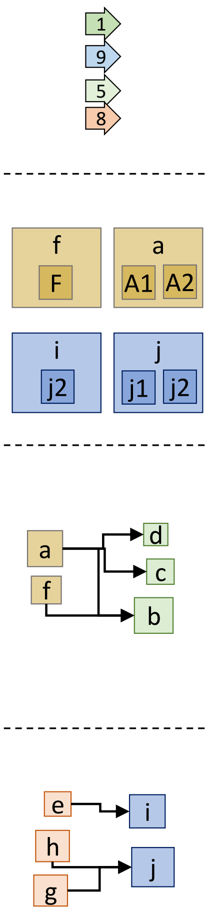


FIGURE 13: TEXTUAL SYNTAX EXCERPT: ARTEFACTS, FRAGMENTS, AND TYPES DECLARATIONS.

```
// Integrity
evidences {
    AIEvidence Evidence_link01 {
       algorithmUsed "AI4All"
       parameters {"platform:/resource/training/posidentifier_202012"}
       executionDate "20201207-123536"
       results .8 .7
       impactedElements ("Transclusion1.link01", sec_A_I, PoS_D)
    },
    AIEvidence Evidence_link02 {
       algorithmUsed "AI4All"
       parameters {"platform:/resource/training/posidentifier_202012"}
       executionDate "20201207-123536"
       results .8 .7
       impactedElements ("Transclusion1.link02", sec_A_II, PoS_C)
    }
 }
referees {
    HumanReferee Sebastien,
    MachineReferee Rd15OUA5RD
}
```

FIGURE 14: TEXTUAL SYNTAX EXCERPT: INTEGRITY ELEMENTS DECLARATIONS.

4 SOFTWARE ARTEFACTS

4.1 ECORE METAMODEL & XTEXT GRAMMAR

The software artifacts we have implemented have been uploaded to the Modelia Git repository [1] and can be accessed in following URL: `https://github.com/modelia/tracea`

Folder `\model` contains the Ecore metamodel, its corresponding Xtext grammar, and a set of well-formedness rules written in OCL. We also added the source for the running example we used to illustrate this report. Here are some adaptations we applied to fit the DSL on Eclipse-EMF [19]:

- We added a class `Tracea` that acts as the root of the metamodel. A `Tracea` has a name and contains the set of `Artefacts`, `Traces`, `Evidences` and `RelationshipTypes` instantiated for cross-reference reuse. *This class is the root containment for Ecore implementation.*
- We slightly relaxed *containment references* to ease the manipulation of references with Xtext. Next iteration on the metamodel should take note of relaxing containment.

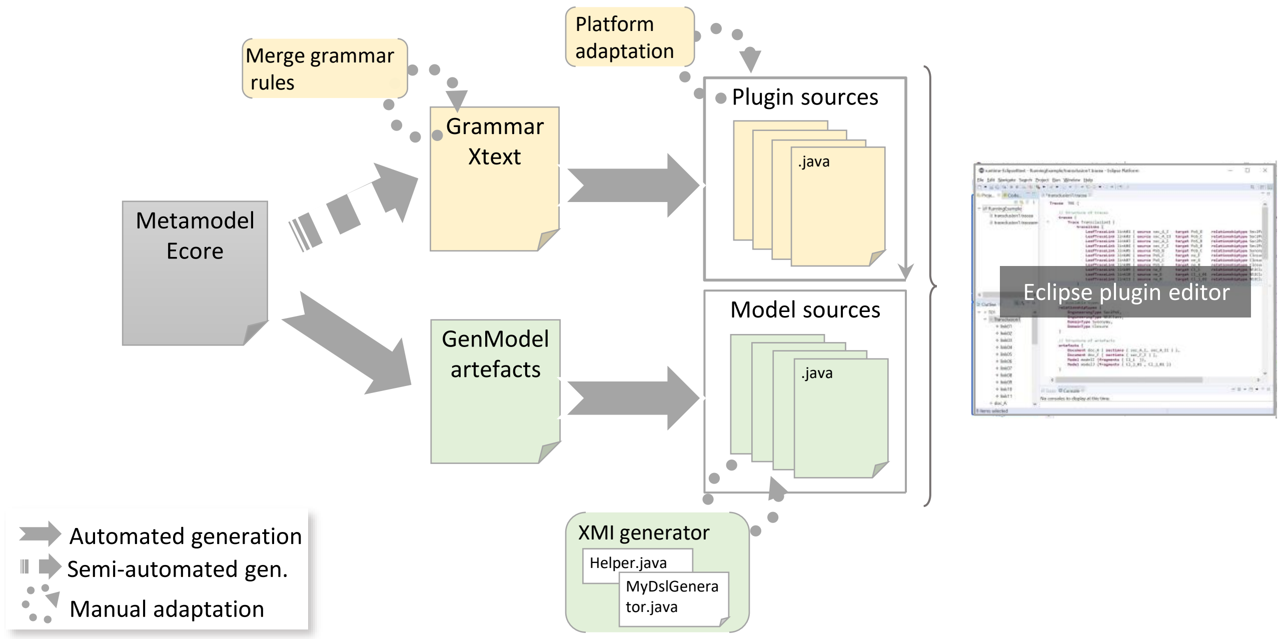


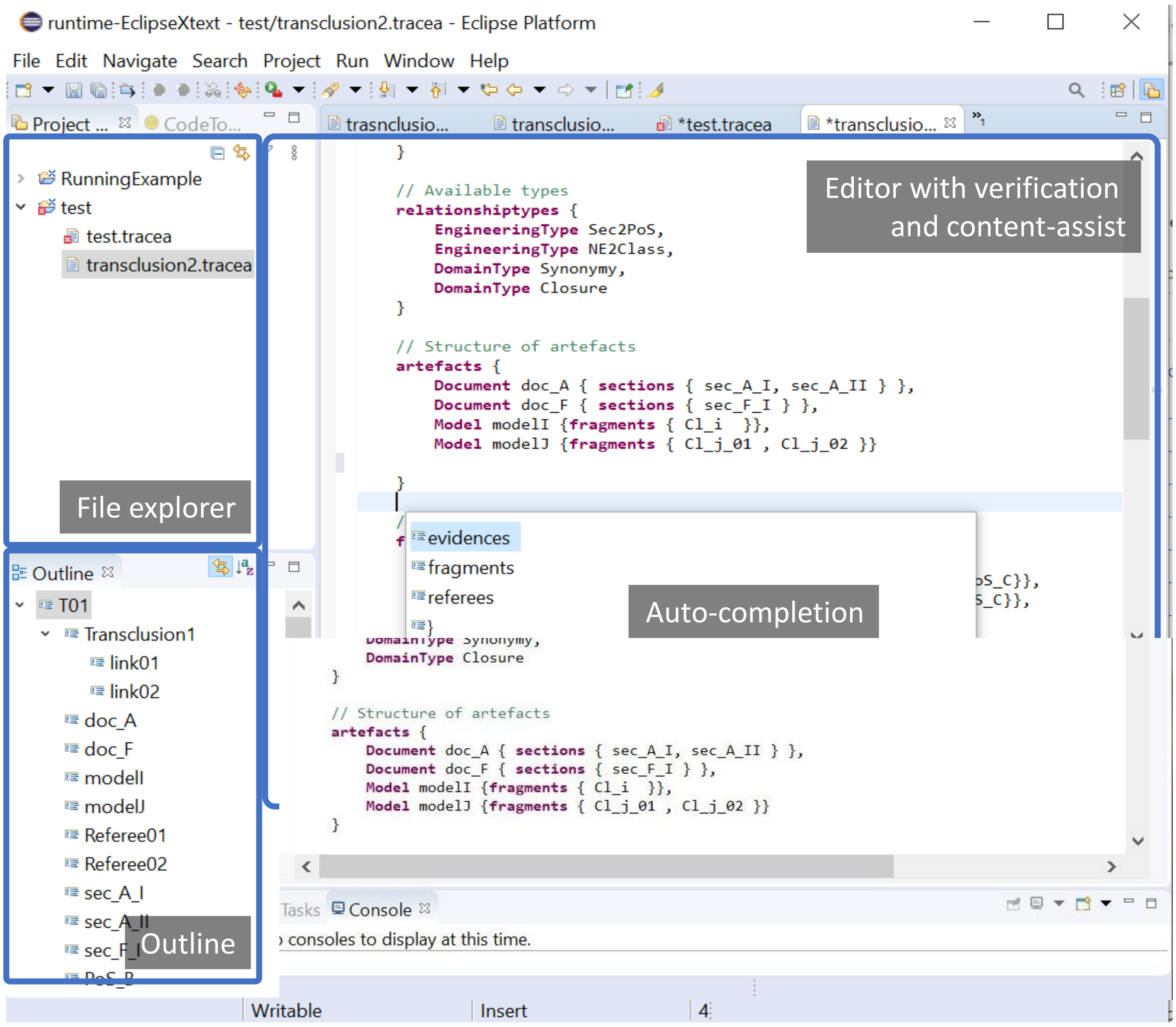
FIGURE 15: XTEXT PLUGIN DEVELOPMENT WORKFLOW

4.2 ECLIPSE XTEXT EDITOR PLUGIN FOR TRACEA

This deliverable contains a concrete syntax written in Xtext [2]. The source of this artifact has been uploaded with the DSL on Modelia Git repository. A set of six projects is part of the execution. In particular, `\Tracea` contains the input artefacts: a metamodel in Ecore, and an Xtext file with its concrete grammar. The other five contain sources generated via Xtext Artefact generator on the EMF platform [19].

Xtext provides support for the parsing of the grammar. We developed a dedicated Eclipse editor for Tracea files and implemented a generator to save Tracea file in XMI format. Figure 15 describes the development of the plugin. EMF uses Ecore metamodel to generate a GenModel file and a set of source artefacts (visible in `\Tracea\model\src`). We modify these files to generate a copy of Tracea files translated into XMI. The modification can be found in `\Tracea\model\xtend-java-edit-for-XMI`. From an Xtext grammar, EMF generates an Eclipse plugin editor. Packages and source artefacts are created in `uoc.som.tracea`, `uoc.som.tracea.ide`, `uoc.som.tracea.test`, `uoc.som.tracea.test.ui`, and `uoc.som.tracea.ui`. Depending on the environment of production, the importation of packages in the Xtext file may vary. The two different versions (with nsURI, or with path) are present at the top of the Xtext. To execute the plugin, a new instance of eclipse must be launched from `uoc.som.tracea`.

Figure 16 shows a screenshot of the Tracea editor. On the left hand, there is a file explorer and an outline (common Eclipse plugin views). On the right hand, an editor that colorizes keywords and spellchecks words. The plugin assists the user with auto-completion and verifies the grammar with the emission of warnings and errors.



```
    }

    // 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 }}

    }
```

```
        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.
```

FIGURE 16: SCREENSHOT OF THE TRACEA EDITOR.

REFERENCES

[1] Edouard R. Batot. Tracea. `https://github.com/modelia/tracea`, 2020.

[2] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. Packt Publishing, 2nd edition, 2016.

[3] Loli Burgueño, Robert Clarisó, Jordi Cabot, Sébastien Gérard, and Antonio Vallecillo. Belief uncertainty in software models. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings*, MiSE '19, page 19–26. IEEE Press, 2019.

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

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

[6] Abel Gómez, Jordi Cabot, and Manuel Wimmer. TemporalEMF: A temporal metamodeling framework. In *Conceptual Modeling*, pages 365–381. Springer International Publishing, 2018.

[7] O. Gotel and A. Finkelstein. Contribution structures [requirements artifacts]. In *Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95)*, pages 100–107, March 1995.

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

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

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

[11] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, 2008.

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

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

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

[15] S. Patel, S. Sihmar, and A. Jatain. A study of hierarchical clustering algorithms. In *2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 537–541, March 2015.

[16] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[17] W. Shen, J. Wang, and J. Han. Entity linking with a knowledge base: Issues, techniques, and solutions. *IEEE Transactions on Knowledge and Data Engineering*, 27(2):443–460, Feb 2015.

[18] George Spanoudakis and Andrea Zisman. Software Traceability: A Roadmap. In *Handbook Of Software Engineering And Knowledge Engineering*, pages 395–428. World Scientific, August 2005.

[19] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

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

[21] Xuchang Zou, Raffaella Settimi, and Jane Cleland-Huang. Improving automated requirements trace retrieval: a study of term-based enhancement methods. *Empirical Software Engineering*, 15(2):119–146, 2010.