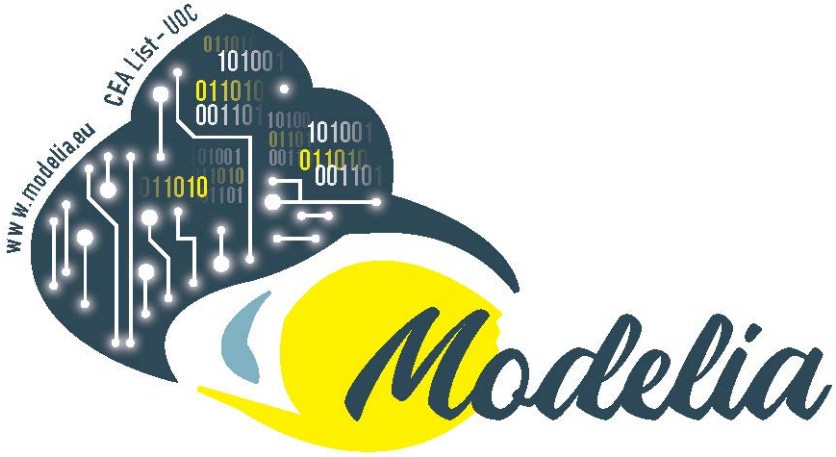
R&I

Universitat Oberta de Catalunya

3RD DELIVERABLE: TRACEABILITY SOLUTIONS – EVALUATION AND EXTENSION



Edouard R. Batot

SOM Research Lab
Av. Carl Friedrich Gauss, 5. Building B3
08860 Castelldefels

SOM

3rd Deliverable:

Traceability solutions

– Evaluation and extension

Contents

List of Acronyms

MDE Model-Driven Engineering

MBSE Model-Based Sofware Engineering

DSML Domain Specific Modelling Language

XMI XML Metadata Interchange

SOM Systems, Software and Models Lab

UOC Fundacio per a la Universitat Oberta de Catalunya

CEA Commissariat à l'énergie atomique et aux énergie alternatives

List of Figures

Listings

1 Introduction

This report documents the work done as part of the third task described in the Convention de l'équipe de recherche commune. It corresponds to Deliverable 3. The core contribution is an evaluation protocol for solutions to traceability (or *tracers*) and its specific application to an existing solution: Capra[1]. The protocol features the evaluation of tracers together with the integration of quality concerns for traceability artefacts.

This document features a protocol to evaluate and extend solutions to traceability (*tracers*). It shows how to integrate trace quality properties, the bottlenecks of the process and its impact on the code. An application onto a specific case shows its usability and provide a first tracer with trace quality for MSBE.

The two first deliverables of the Trace*a* project constitute the conceptual basis behind this work. The first deliverable, a survey-driven feature model for traceability, shows the general absence of consideration toward trace quality in existing approaches [1]. The second deliverable presents Trace*a*, our modular metamodel for traceability [2]. The complete Trace*a* metamodel is available in a paper we submitted to the international working conference on Exploring Modeling Methods for Systems Analysis and Development (EMMSAD 2021) and available on the Modelia's Git repository [3]. In this third deliverable, we integrate parts of Trace*a* to extend and complete Capra.

More specifically, in the next section, we propose a protocol to evaluate the relevance and the robustness of solutions to traceability. This section also sketches steps to extend tracers with quality aspects for trace links. Section 3, depicts the application of the protocol to Capra through our manual investigation. In section 4, we detail the integration of Trace*a* into Capra and the impact of the consecutive changes. In Section 5, we show the main conceptual and technical limitations of Capra before we conclude in Section 6. The edited software product is available in Section 7.

[1] `https://projects.eclipse.org/projects/modeling.capra`

2 Evaluation protocol for traceability solutions

The quality of traceability solutions lies on five main pillars : (1) The customization of the tool to adapt to different systems[2], (2) the level of automation of the identification of traces, (3) the means of visualisation and retrieval of trace links, (4) the means of persistence and additional operations available, and (5) level of quality assessment of the tracing constituents. In this section, we present these as five independent dimensions for the evaluation of traceability solutions.

This section presents a protocol to evaluate tracers. The main focus in the consideration of the trace artefacts themselves, their customization, identification, visualization, persistence and quality assessment. It also sketches a protocol to integrate Trace*a*'s quality definition into tracers.

Fig. 1 schematizes the overall traceability infrastructure. It shows the three main components: the traces and their constituents, the environment in which it operates, and the artefacts it targets on (a) system (of systems). The figure shows where the dimensions for evaluation apply.

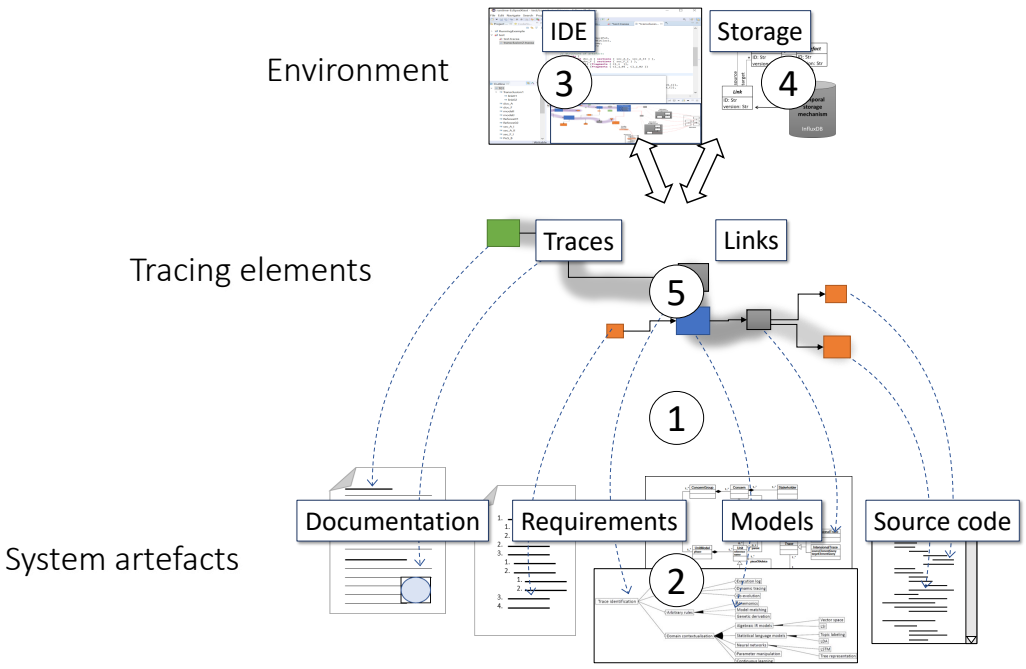


FIGURE 1: Overview of the traceability infrastructure with localisation of the dimensions for evaluation

[2]We call *system* or *software system* interchangeably here, considering a software system an abstract representation of a (software) system.

2.1 Evaluation of traceability solutions

In more details, we define five dimensions to evaluate a solution to traceability (as summarized in Table 1).

① **Customization**	- Artefact typing & Granularity - Relationship typing & Cardinality - Trace structure
② **Identification**	- Automation level - Extra features
③ **Visualization & Retrieval**	- Textual, Graphical, Matrix-based representations - Other types of representation - Querying existing traces
④ **Persistence & Edition**	- Persistence format - Operations on traces and links available
⑤ **Quality assessment**	- Consistency - Confidence - Explainability

TABLE 1: Five pillars for traceability evaluation.

1 - CUSTOMIZATION To evaluate the potential a tool has to address a specific purpose, **customizability** must be evaluated with regard to the kind of artefacts and the kind of links it offers to manipulate[3]. A generic tooling must allow an *ad hoc* definition of types to adapt to different uses in different application domains [11].

2 - IDENTIFICATION What are the means available for the identification of trace links? This task can be automated to different levels. Links can be identified manually, which usually depends on the integration of the tool into the system's environment. Links can also be identified automatically using rules and/or AI techniques. Moreover, there exists explicit links in the syntax trees of many languages. These can be automatically derived, included or used for visualization purpose as an extra feature (or known neighborhood).

3 - VISUALIZATION & RETRIEVAL Once traces are identified, they can be represented in form that fit the needs of the user. *E.g.*, either textually, or diagramatically, in spreadsheet or even in matrix, revealing their structure in a (more or less) legible way, and editable to a certain degree. Matrix-based representations show a broader aspect of the traces in a more "conventional" way that some favor [9]. For example, the matrix view makes it easy to see if all requirements are associated to test cases.

[3]We call "trace links", "connections", "relationships", simply *links* in this document, as opposed to *trace* which are entities composed with (successive) links.

There might as well be a great number of traces identified, for various purposes, and a proper tooling to retrieve traces of interest is important. This can be done, for example, with a query-based mechanism.

4 - PERSISTENCE & EDITION The format in which traces are serialized matters. It can be embedded in the artefacts themselves, or external to the software through independent traceability artefacts. These can be store as model elements, SQL scripts, or XML sheets, and so on depending on traceability requirements (*e.g.,* what is most important: expressiveness, efficiency, or interoperability?).

Traces are complex entities and a robust tooling should also offer robust operations to handle them with precision and efficacy.

5 - QUALITY ASSESSMENT Finally, how is the quality of the trace and its constituent assessed? Traces suffer gradual decay, depending on the degree of versatility of the system, they might not be kept up-to-date, and their consistency with the system's artefacts may decline. This indicates that traces might suffer some uncertainty. Is the confidence about the degree of existence of a trace evaluated? Is it possible to record and reuse it? If a trace has been identified automatically, traces should refer to evidences about the rationale behind their identification. Is the tool offering any means to express these considerations?

2.2 Extension for trace quality

We foresee a recurring lack in the fulfilment of the last dimension "5 - Quality assessment". All approaches we have seen so far are oblivious of this aspect [2, 4]. We propose to address this limitation through an extension protocol that integrates trace quality into traceability solutions. Fig. 2 shows an excerpt of Trace*a* dedicated to the quality of traces and links. This excerpt features confidence values associated with evidences - when they exist. The root element for tracer must also refine `TracingElement` to advocate for `Agent` responsibility in the tracing process. A detailed review of the conceptual modeling beyond trace quality classes imported by Trace*a* is provided in [5].

1 - LOCALISATION First of all, we need to locate the core conceptual representation of trace link. Is it a model that generates some low level code? Is it rendered directly in low level code such as Java or C? Is it explicitly modularised or is it part of a bigger picture?

The extension point for tracers to integrate Trace*a* lies in the left-hand classes `Trace` and `TraceLink` of Fig. 2. Both are susceptible to give a confidence value, supported by evidences if any.

2.2 Extension for trace quality

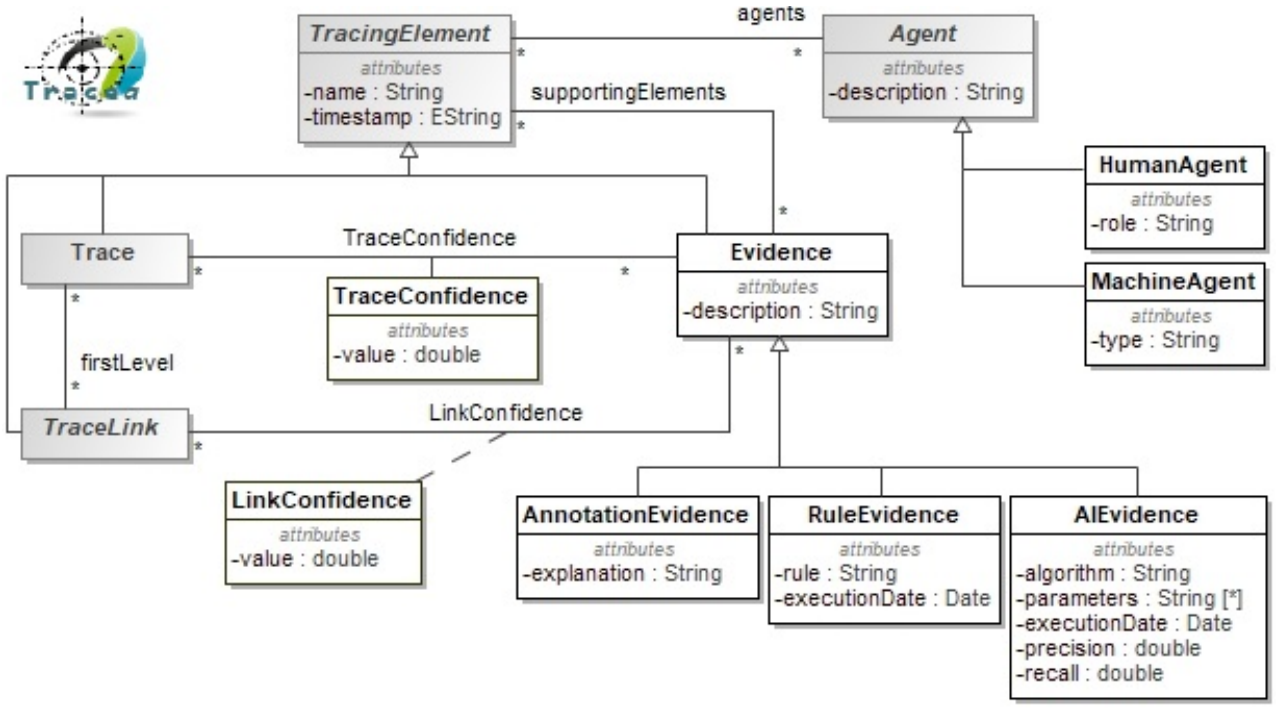


FIGURE 2: Excerpt of Tracea dedicated to the quality of traces and links.

2 - INJECTION Then, Trace*a* quality properties must be added to the definition. This may be done through a modification of the metamodel (in the case of low code generation) or a direct edition of low level code related to links and traces. In both cases, Trace*a* must endorse a translation into the language/technology of the targeted tracer.

3 - INTERFACING Once quality properties have been added to the tracer, it must be propagated through the different packages/modules of the tool. This implies understanding the general architecture of the tracer to be able to add quality properties as a core feature without *breaking stuff*[4].

4 - CHANGE IMPACT Finally, a change impact analysis must be operated to ensure the changes do not impact unexpected elements of the tracer. This must be done at the specification, design, source code, and test level.

[4]Facebook's mantra for developers has long been "Move Fast and Break Things." After decade long polemic over its legal implication, the Menlo Park company has stepped down. Facebook is now embracing the motto "Move Fast With Stable Infra."

3 Evaluation of a traceability solution: Capra

Capra is a promising generic solution to model-based software traceability [7, 11]. Yet, as we will see, it suffers an obvious lack of means to consider the quality of traces. In this section, after a brief introduction to Capra, we detail the application of the protocol introduced in Section 2.1 on that specific solution. We detail each of the five steps and summarize the results in Table 2 to establish whether, or to what cost, Capra would fit a consequent industrial exploitation.

This section depicts how the protocol from Section 2.1 has been applied to evaluate Capra. The adaptativeness of the tool through customization and visualization features is put forward ; identification and quality assessment remain slightly back.

QUICK PRESENTATION OF CAPRA Capra is a tool and a research project aiming at generic software traceability [7, 11]. Its initiators rose the ability to trace as a noticeably important feature to integrate into the realm of model-driven engineering [8]. Scientific publications show that strong traceability has been long due in the field and Capra's authors distinguish themselves from other approaches with an important focus on customizability and visualization of tracing artefacts [6, 10]. The overall goal of Capra is to allow the linking of elements from different domain specific modeling languages.

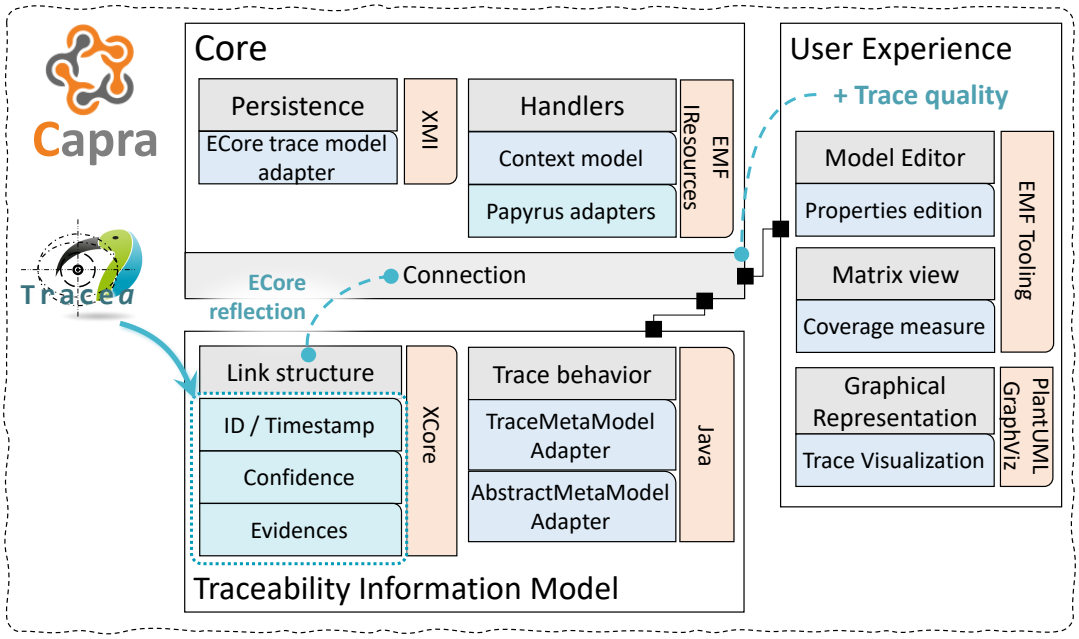


FIGURE 3: Architecture of Capra

See Fig. 3 for a detailed view of Capra architecture. The implementation of the tracer is based on the Eclipse Modeling Framework [13] hosted open on Git [12]. It consists of an Eclipse plugin that carries out the requirements an industrial partner mandates. There is a core package, which grows into new handlers to support more and more domain specific languages and usages[5]. A package is dedicated to the customization of the traceability information model (or trace model), and the UI is based on Eclipse's ECore model editor and PlantUML for diagrammatic representations.

The maintenance of the tool is ensured mainly by the authors of the publications. They are quick to answer questions and comments.

3.1 Traceability customization

Capra allows a custom definition of the artefacts' and relationships' types. The customization of artefacts and links is editable through an XCore plugin[6]. Links can be defined following a hierarchical inheritance and allowing structural features (reference and attributes)[7].

Capra is an EMF plugin and benefits from the artefacts used by the running instance of the platform. An artefact is defined through the resource change management plugin mechanism of EMF infrastructure. The wrapping of the artefacts uses another XCore plugin. It allows the definition of other attributes for artefacts: `org.eclipse.capra.generic.artefactmodel`. The adapters, handlers, and helpers for the EMF artefacts must be redefined as Java polymorphic overriding projects (derived from `org.eclipse.capra.generic.core`). Capra redefines wrappers for 15 languages or standards such as UML, BPMN, Papyrus, Mylyn, Java, C++ Office, to name a few[8].

By default there is one generic *RelatedTo* possible kind of link. For each link, there must be one and only one source and one or more target artefacts (target cardinality can be changed). The types of origin and target artefacts can be constrained.

In the default implementation, links are part of a `TraceModel` that *connects* between them. There is no explicit trace structure but a *behavior* defined in Java. If an artefact is part of the target of a link and the origin of another, the trace connects (see the Section 3.3 on visualization). This implementation (`org.eclipse.capra.generic.GenericTraceModel`) must be transferred or adapted (Java file, no documentation) to be adapted to a new structure. There is no test coverage for potential alterations to the original implementation.

[5] To quote but one example, Capra is ready to use with Papyrus models and elements.
[6] More details can be found at: `https://wiki.eclipse.org/Capra/CustomTraceabilityMetaModel`
[7] There is no guaranty on the level of implementation of the more complex features XCore allows.
[8] There is no documentation relating the means to apply such redefinition - neither can be found any tips on the potential impact such changes may initiate.

3.2 Links identification

Identifying links in Capra is done manually. Any element actionable in an Eclipse environment can be selected to take part of a trace. A context menu reacts to elements and offers to add to the "Traceability selection", or drag-n-drop the element into the `Trace Creation`[9] view (Fig. 4). (Respective adapters are required, see Section 3.1.) There is no automation for the identification of relationships, but the UML dependencies can be viewed in the `Capra PlantUML Viewer`. UML relations are not part of the serialized of the trace links. They are derived for (visual) convenience and they need to be manually recorded to be serialized.

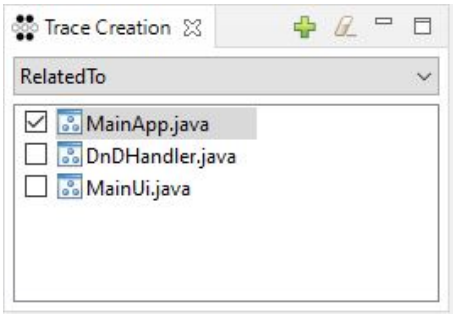


FIGURE 4: Capra's Trace Creation view

3.3 Trace visualization & Retrieval

Capra's prime feature is a multi-paradigm visualization support with textual, graphical and matrix representations. The second, based on PlantUML/Graphviz, supports developers in change impact analysis ; the matrix representation facilitates the evaluation of test coverage. Mixing the two allows an in depth safety analysis - Capra's authors next publication will show this last point[10].

AUGMENTED TEXT REPRESENTATION OF LINKS Traces in Capra are serialized into XMI Ecore models. This is convenient since Eclipse offers a `Reflexive Ecore Model Editor` to browse and edit such resources. Traces themselves cannot be apprehended though, if not to a great cost, because this view does not show the succession of trace links. Fig. 5 shows a screenshot of this view. As can be seen, a name composed with the name of the artefacts targeted is the only information available - together with the elements properties in the `Properties` view associated. Instances of the wrappers for the artefacts used by the trace links are stored in the same manner in another XMI file.

[9]Names are subject to change.
[10]From an interview with the main Git contributor and author.

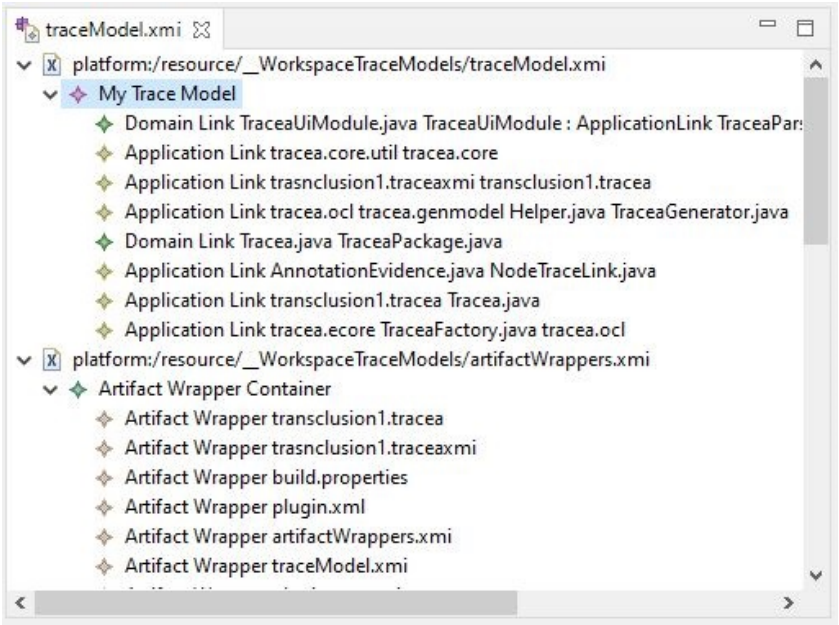


FIGURE 5: Links and artefact wrappers, browsed with Eclipse Ecore Model editor

GRAPHICAL REPRESENTATION OF TRACES To ease the visualization, Capra generates a graphical description of traces in GraphViz[11] and uses PlantUML[12] to plot them in an Eclipse view. Figure 6 shows a capture of the view. This view shows the links associated to the element selected in the Eclipse IDE. These links are the one identified with Capra (the trace links) or are derived from the neighborhood of the element selected. In the case of a Java class, a type hierarchy is provided. The view is configurable. The type of links, as well as the depth of transitivity can be changed through the Eclipse interface.

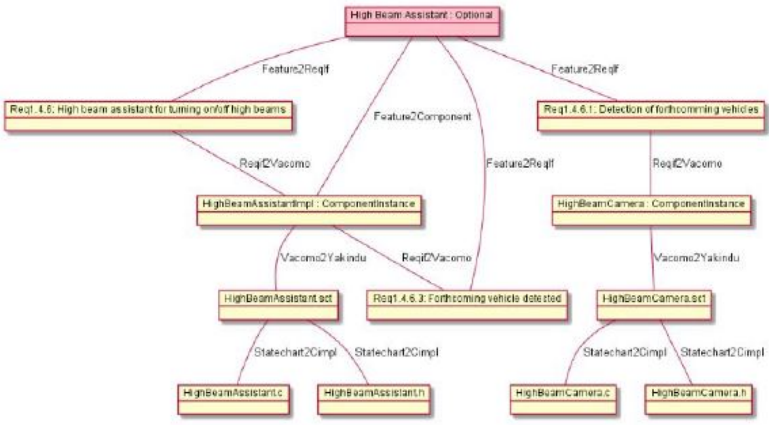


FIGURE 6: Trace links sequences are plotted graphically in Capra PlantUML Viewer

[11] https://graphviz.org/
[12] https://plantuml.com/

MATRIX-BASED REPRESENTATION OF TRACING Capra offers an association matrix view, as can be seen in Fig. 7. This view shows links from a high-level perspective and is most useful for coverage measurement. It can be exported in the Excel (xls) format.

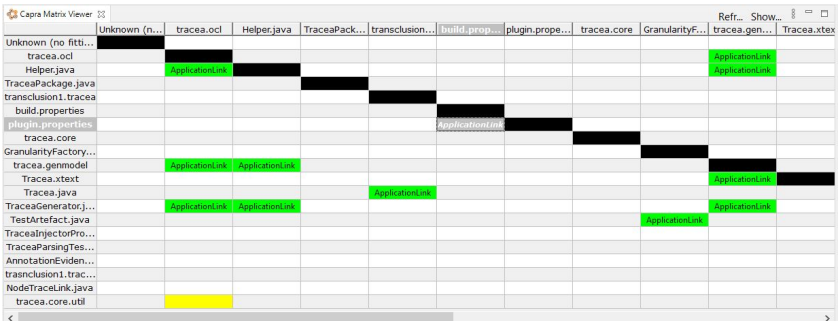


FIGURE 7: Interrelations can be viewed in the matrix-based representation

LINK RETRIEVAL There is no mechanism to retrieve automatically trace links. Yet, the use of Viatra-IncQuery[13] is a strong bet to address this limitation for it is tailored to XMI Ecore resources and offers many features for querying and generating Ecore models.

3.4 Trace persistence & Edition

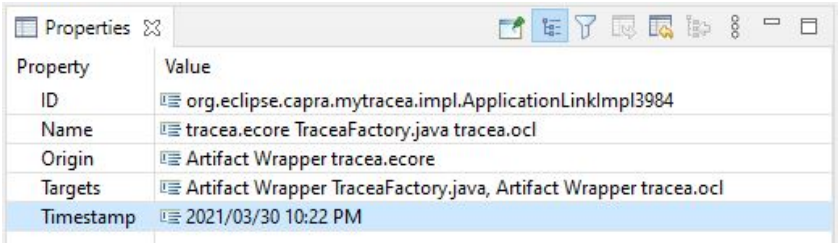


FIGURE 8: Links are viewed and edited with Eclipse Properties view

Traces are serialized into an Ecore model, stored in an XMI file. Artefacts wrappers are also stored in XMI. The `Reflexive Ecore Model Editor` (together with the `Properties` view, see Fig. 8) can be used to edit the information but **the consistency of other views is not guarantied.** Eclipse must be closed and reopened. The name of the files appear in the project list, but their name cannot be edited. **It is highly recommended to not modify the files, nor delete them.** The default `PlantUML viewer` does not offers any means to interact with the trace.

[13] `https://www.eclipse.org/viatra/`

3.5 Trace quality evaluation

Capra checks the consistency of traces when artefacts are modified or deleted through the Eclipse platform. Changes in the software elements triggers a validity check on the links these elements take part in (as source or target). As a consequence, the links are tagged with a warning. There is no fix proposed.

Confidence is not quantified and there is no record of the facts or evidences about the existence of a link.

3.6 Conclusion

Table 2 summarizes the evaluation of Capra, and Fig. 9 illustrates the summary with regard to the traceability infrastructure overview. As can be seen, the tool benefits a strong customization potential. On the one hand, it is easy to design wrappers for new artefacts with Eclipse `IResource` redefinition, and the definition of link types is very expressive thanks to XCore. On the other hand, if the tool lacks some automated mechanisms to identify traces, the visualization of both specific links and derived UML relations is very powerful. Capra is currently used in three use cases: Change Impact Analysis, Coverage analysis, and Safety analysis, with a publication to come on these matters.

On a bad note, the absence of means to edit safely trace links is a major impediment to the use of Capra in an industrial context. Discussing with the authors on the matter showed there is no plan to address this issue. They do not consider the use of the XMI editor as a convenient mean to edit traces.

The tool also suffers a lack of trace quality assessment. There is one feature related to quality in Capra which consists in a consistency check that triggers warnings when resources targeted by traces are modified or deleted in the Eclipse environment. Yet, there is no evaluation of any kind of confidence level and less explainability resources. This is a recurrent limitations that most if not all traceability solutions suffer [4, 5].

As a conclusion, Capra is a rather good tool for traceability in terms of adaptativeness – and that is a very important concern to tracer design. The quality of trace links is not considered, as expected – but we show in the next section how to extend Capra with Trace*a* to circumcise this limitation.

Customization	- Custom artefact and link types easily defined in XCore language - There exists artefact wrappers for more than 15 modeling languages and standards - Default links are single origin multi target (1-- *) (configurable) - Trace structure is derived from links
Identification	- Manual identification anywhere in Eclipse IDE (if respective artefacts types are defined) - No automated identification - PlantUML shows internal structure of UML (and Java) elements

Visualization & Retrieval	- Textual visualization with Eclipse XMI Reflexive Editor
	- Graphical representation generated with GraphViz/PlantUML (no interaction)
	  - Rendered types and transitivity length configurable
	- Matrix-based representation
	  - Rendered types configurable, no transitivity
	- No retrieval or query of existing trace links

Persistence & Edition	- XMI synthesis (and GraphViz for graphical representation) - Singleton syndrome - Edition of trace links thanks to Eclipse integration causes risk of inconsistency between the trace instance and the different views
Quality assessment	- Consistency: when an artefact is modified or deleted, the referring links are tagged - Confidence: Ø - Explainability: Ø

TABLE 2: Summary of the evaluation of Capra

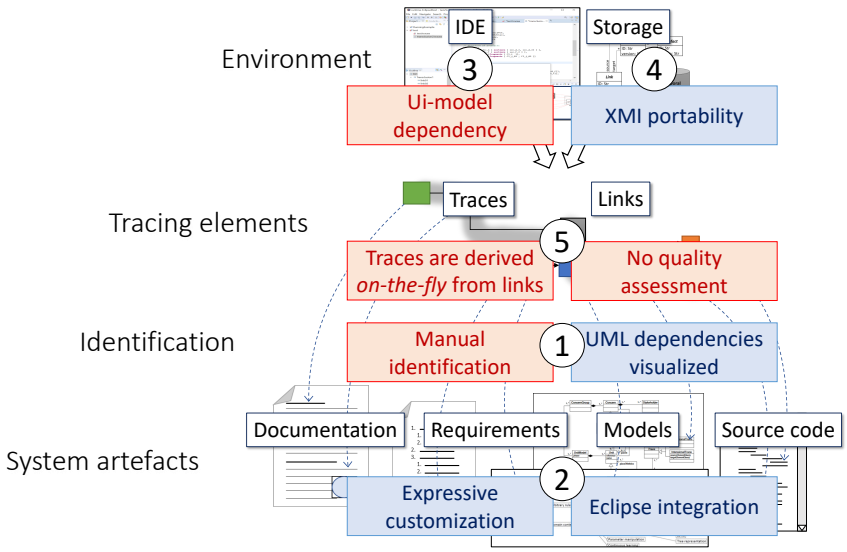


FIGURE 9: Summary of the evaluation of Capra

4 Integration of Trace*a* in the Capra solution

This section depicts how the protocol from Section 2.2 has been applied to extend Capra. It describes how to foster trace quality, details the new features and their benefits, the nature of changes made, as well as the potential impact such changes may propagate.

As we have seen, Capra does not offer any means to address the variation in quality of trace artefacts. This is a recurring limitation for which we propose a meta-description and sketched a protocol to integrate it.

In this section the module from Trace*a* that tackles quality issue is integrated into Capra following the protocol presented in Section 2.2. We first introduce the integration of trace quality into Capra's traceability information metamodel (the localisation of conceptual changes). Then we show the modifications we carried out: the injection of quality features into the core package of Capra and its propagation to the other modules. Finally, we uncover the new features allowed by these changes (UI augmentation). The implementation of such changes impacts the quality of the overall Capra solution. We depict these concerns as well as potential future work at the end of the section.

4.1 Add trace quality to Capra's metamodel

We address the lack of quality assessment of trace elements by hacking the customization mechanism of Capra. We integrate Trace*a*, our metamodel for traceability [5], into Capra as a custom metamodel[14]. To do so, we adapt Trace*a* to fit with Capra's requirements and translate it as an XCore project.

4.1.1 Trace model, relationships, and agency

As can be seen in Listing 1, at the root of the Ecore model, a trace model is defined with links and agents. All concepts (except for the model) derive from `TracingElement`. They have a unique ID and a time stamp. They may also refer to which agent is responsible for their creation. An `Agent` may be a human being (`HumanAgent`) or a piece of machinery (`MachineAgent`). The top level type of the relationships is `RelatedTo` – a generic trace link that connects one origin artefact to potentially many targets. These implementation decisions follow the genericity requirement Capra asks for. Following the declaration of *RelatedTo* any other kind of relationship can be easily derived.

[14] As explained in `https://wiki.eclipse.org/Capra/CustomTraceabilityMetaModel`

```
class MyTraceModel {
  contains TraceLink [0..*] links
  contains Agent [0..*] agents
}

abstract class TracingElement {
  derived String ID get { EcoreUtil.generateUUID(); }
  String name
  String timestamp
  refers Agent agent
}

abstract class Agent extends TracingElement {}

class HumanAgent extends Agent { String role }
class MachineAgent extends Agent { String machineType }

abstract class TraceLink extends TracingElement {
  String name
  double confidenceValue
  contains Confidence [0..1] confidence
}

class RelatedTo extends TraceLink {
  refers EObject [1] origin
  refers EObject [1..*] targets
}
```

Listing 1: Tracing element, trace model, and trace links

4.1.2 Trace confidence and explainability

Listing 2, represents the `Confidence` value (i.e. an estimation of the existence of the link). Trace*a* defines classes for the different kinds of evidences that may testify the value of the confidence. An `Evidence` can be a textual annotation (`Annotation-Evidence`), or a `RuleEvidence` when a link is created automatically, or an `AIEvidence` when a learning algorithm is used to identify the link. These latter possess attributes to reproduce their execution (*e.g.*, the rule, a certain kind of algorithm, the data set used for training).

```
class Confidence extends TracingElement {
  contains Evidence [0..1] evidence
  double value
}
```

```
abstract class Evidence  extends TracingElement {
  String description
  refers TracingElement [0..*] supportingElements
}

class AIEvidence extends Evidence {
  String algorithm
  String dataSet
  int executionDate
  double precision
  double recall
}

class RuleEvidence extends Evidence {
  String rule
  int executionDate
}

class AnnotationEvidence extends Evidence {
  String explanation
}
```

Listing 2: Confidence and evidences for trustable traceability

4.1.3 Code change requirements

To integrate the new link types and link attributes to Capra, interfaces must be realized. Fig. 10 shows the relation of these interfaces, the trace model, and the relationships.

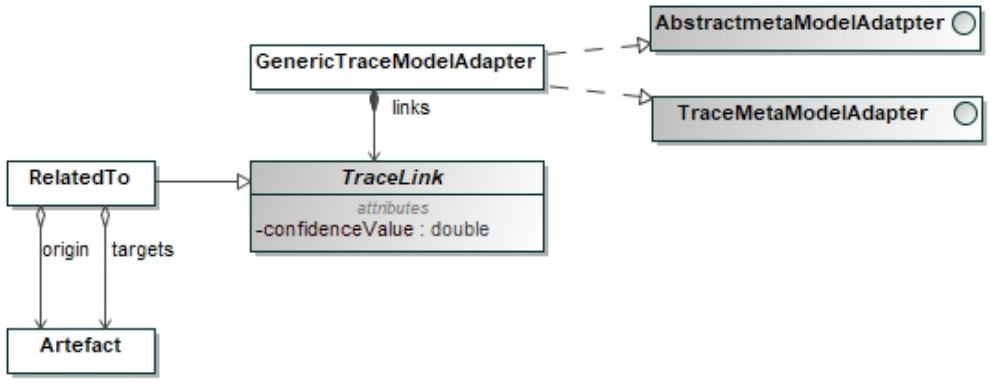


FIGURE 10: Excerpt of the API realization for the definition of trace model and links

The interface `org.eclipse.capra.generic.tracemodel.TraceMetaModelAdapter` must be adapted (see Listing 3) to define generic methods to browse links. The links with their origins and targets are wrapped into `Connection` to bridge with other packages through ECore reflexion API. Everything is configurable. As long as new link types derive from `TraceLink`, they will be augmented with quality attributes (*i.e.* confidence and evidence).

```
EObject createModel();
Collection<EClass> getAvailableTraceTypes(List<EObject> selection);
EObject createTrace(EClass traceType, EObject traceModel, List<EObject> origins,
    List<EObject> targets);
boolean isThereATraceBetween(EObject origin, EObject target, EObject traceModel);
List<Connection> getConnectedElements(EObject element, EObject traceModel,
    List<String> traceLinkTypes);
List<Connection> getTransitivelyConnectedElements(EObject element,
    EObject traceModel, List<String> traceLinkTypes, int transitivityDepth);
List<Connection> getAllTraceLinks(EObject traceModel);
```

Listing 3: TraceMetaModelAdapter interface

In the same manner, the interface `org.eclipse.capra.generic.tracemodel.AbstractMetaModelAdapter` offers an interface to define how *internal traces* (*i.e.* links between internal parts of artefacts) must be handled (see Listing 4). The default `org.eclipse.generic.tracemodel.GenericMetaModelAdapter` offers a basic adaptation to these interfaces.

```
void deleteTrace(List<Connection> toDelete, EObject traceModel);
List<Connection> getInternalElements(EObject element, EObject traceModel,
    List<String> traceLinkTypes);
List<Connection> getInternalElementsTransitive(EObject element,
    EObject traceModel, List<String> traceLinkTypes, int transitivityDepth);
boolean isThereAnInternalTraceBetween(EObject first, EObject second);
```

Listing 4: AbstractMetaModelAdapter interface

4.2 Extend Capra's core interface

As previously stated, we add a top level link type (`TraceLink`) to inject the confidence of trace links in Capra. We access the newly added attributes through a reflexive exploration of the trace model structure with *EMF Reflexion* as can be seen in the excerpt in Listing 5, from `org.eclipse.capra.core.EMFHelper`.

```
public static double getConfidenceValue(EObject tlink) {
    EObject confidenceEO = (EObject)tlink.eGet(
            getEStructuralFeatureByName(tlink, "confidence"));
    if(confidenceEO == null)
        return DEFAULT_CONFIDENCE;
```

```
6       return (double)confidenceEO.eGet(
7               getEStructuralFeatureByName(confidenceEO, "value"));
8   }
9
10  public static EStructuralFeature getEStructuralFeatureByName
11                                          (EObject eo, String esfName) {
12      for (EStructuralFeature esf : eo.eClass().getEAllStructuralFeatures())
13          if(esf.getName().equals(esfName))
14              return esf;
15      return null;
16  }
```

Listing 5: EMFHelper: reflection excerpt for the integration of a confidence value into Capra's `Connection` interface

4.3 Augment Capra's user experience

Capra is conceived with the goal to add awareness to its users. Its prime feature is the visualization with textual, graphical, and matrix representations. We added features to show and use the confidence in Capra and used it to parameter its user interface, both in textual, graphical, and matrix views.

4.3.1 Interactive textual edition

Based on Eclipse XMI Ecore Model Reflexive Editor, changes in the trace model were integrated automatically. There must be more effort put on insuring the model remains consistent with the different view. Fig. 11 shows the editor view used for link edition. There, we can see the structure of a link, its confidence value, the evidence associated and potential agents responsible for these elements.**Edition makes the overall application unstable.**

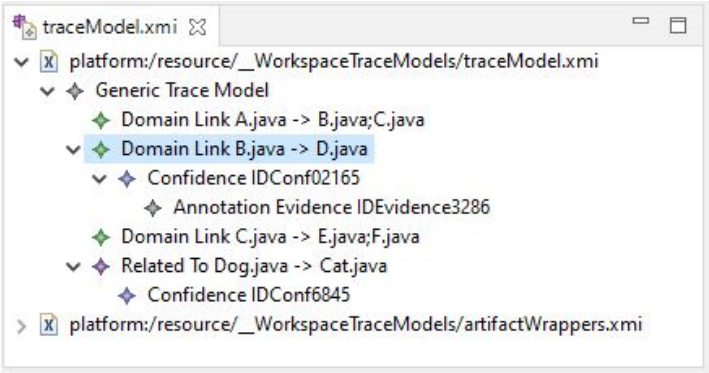


FIGURE 11: Link structure representation and edition in Eclipse Ecore Model Reflective Editor

4.3.2 Matrix UI

We modified the UI of the association matrix. We use the confidence value to decide weather a cell is green or red. The threshold between the two can be changed through Eclipse interface. Fig. 12 shows an example of the new UI. Red cells reveals ”uncertain” links.

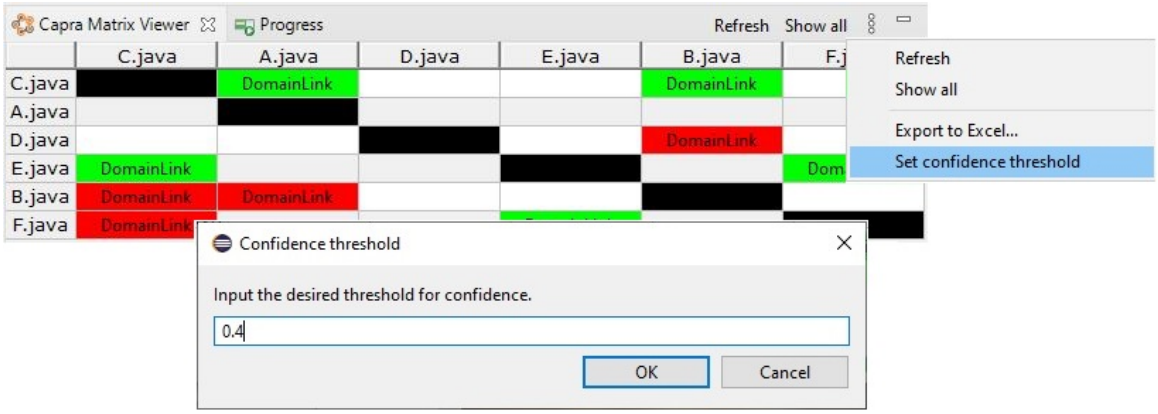


FIGURE 12: Representation of uncertainty in matrix view

4.3.3 Plant UML

We modified the UI of the arrows in PlantUML to show the confidence of a link (when it is not 1.0). A threshold value for confidence can be set to decide weather a link is shown when transitivity view is *on*.

4.4 Change impact evaluation

The *ad hoc* reflexive exploration that interfaces between the trace metamodel and the core package is the main bottleneck for later maintenance. Its consistency with the structure of a link and the *EObject* wrappers (`Connection`) that Capra manipulates must be tested when the trace metamodel is modified. The choice of this solution is due to the customization mechanism. To allow the redefinition of new types, we can only use the *ExtensionPoint* defined (see Adding a custom Traceability Metamodel[15]).

Changes made for the propagation of confidence and its graphical representations of confidence have been posted in a push request to integrate Trace*a* into Capra's branding and thus benefit from its maintenance. The push has been reduced to the minimal introduction of the confidence value for relationships due to the project requirement on genericity.

[15] `https://wiki.eclipse.org/Capra/CustomTraceabilityMetaModel`

4.5 Future work

We showed the adaptability of Trace*a* and the value of its concepts. Its integration into Capra opens the door to new potential features.

HEAT MAP FOR MATRIX VIEW The matrix representation for tracing are now always considering a Manichean existence / non-existence perception of links. We plan to exploit the confidence value to augment the perception of traceability through a heat map relating confidence levels.

UNCERTAINTY REPRESENTATION IN GRAPHICAL VIEW The PlantUML viewer ca be used to plot confidence in other ways than textual or matrix-al. We already integrated a threshold to show only "certain" links, and this research direction could lead to more specific tracing detail .

PLANTUML AND MATRIX VIEWER EDITION OF TRACES The direct edition of traces and links in the different representations would be a important improvement in the usability of Capra.

To operationalize Capra into an industrial environment would require an upgrade of Capra's main design architecture. We develop on Capra's limitation in the next section.

5 Limitations of the Capra solution

This section shows Capra's limitations in term of software design. The singleton syndrome and the absence of link editor being the most salient limitations to plan an industrial exploitation of the tool.

The ability of Capra to represent traces and customize relationships and artefacts is promising. Nonetheless, the tool implementation suffers important limitations. In this section, we present architecture flaws that would restrain an industrial exploitation of Capra. We show the technical limitations at design and code level that need to be addressed.

5.1 Singleton syndrome

The main limitation of Capra lies in the design choices made to implement traces. Its architecture implies that using Capra remains only a one shot deal. There is one file to store links. "Traces", or better said *the trace*, is regenerated from this file at startup. If the user wants to create different *traces*, s.he will have to start a new instance of Capra with a new XMI file. This is what we call the *singleton syndrome* where a software (OO) is designed around a unique (often static) instance of its main concept. Capra follows this pattern.

To address this limitation would require a general lift of the manipulation and exploration of traces and links. It would also require modifying high-level architecture of the tool. A report explaining the limitation has been sent to the shareholders of Capra. This issue is under current inspection by the development team.

5.2 Single step trace links

The next limitation is that Capra does not offer any opportunity to synthesized the trace structure. The trace is derived from the trace links on demand for visualization only. The way a trace is derived can be configured freely in the trace model implementation (see `org.eclipse.capra.TraceMetaModelAdapter`) but can only describe one trace due to the previous limitation.

A solution could be buffering derived traces to avoid reconstructing them to often. Otherwise, without optimization, the tool will hardly scale.

5.3 Trace edition shortfall

We use the XMI Reflexive Model Editor to edit trace links and attribute them confidence value and properties. This shall be improved in the future because for now, there is no assertion

that the over views on the trace instance remain consistent between one another. In other wother words, modifying trace links through this editor implies to restart Eclipse to take these changing in consideration.

The development team is aware of this limitation but does not plan to address it, considering that the edition of trace links is of no use.

5.4 Naming convention and general quality

The use of a `Connection` adapter for links is sometimes misleading. In the PlantUML viewer, the trace links show their corresponding origins and targets whereas EMF elements show their internal structure. The later is defined in the wrappers of the artefact model as described in Section 3.

"Trace", "Link", "Connection", "Connections", "TraceLink" sometimes collide in method and attribute signatures. *E.g.*, In `AbstractTraceMetamodel.createTrace` and `AbstractTraceMetamodel.deleteTrace` "trace" entities are actually connections. Which is fundamentally misleading.

There can be found cut-n-pastes as well which we should be careful about, specifically in the generic trace model (`org.eclipse.capra.generic.tracemodel`), and the generation of PlantUML diagrams (in `org.eclipse.capra.ui.plantuml.DiagramTextProviderHandler`).

6 Conclusion

In this document we propose a protocol to evaluate solutions to traceability. Five main dimensions are considered: customizability, identification means, visualization and retrieval features, persistence and edition of traces, and trace quality considerations. Capra responded positively to most of the requirements we defined - excepted the quality dimension. We also sketch a protocol to integrate Trace*a* in order to extend solutions to traceability suffering such limitation (related to trace quality) and we apply it to Capra. The overall result is promising, more since Capra is ready for Papyrus development. The tool is ready, with some tuning improvement, for case studies and empirical experiments. Nonetheless, Capra shows design flaws that would need be addressed to target an industrial environment. If the singleton syndrome is being addressed by the development team, the edition of trace artefact is not on plan and this is a serious impediment to Capra's usability.

7 Software artefacts

The software artifacts we have implemented have been uploaded to the Modelia Git repository and can be accessed in following URL: `https://github.com/modelia/tracea/` (folder `capra-integration`).

The repository contains an edited copy of Capra (v8.0.2) with a custom traceability metamodel, the extension of Connection and EMFHelper from `org.eclipse.capra.core.adapters` and `org.eclipse.capra.core.helpers`, and UI patches to use confidence value in the viewers.

REFERENCES

[1] Edouard R. Batot. First deliverable: Survey of traceability techniques with a focus on their applications in ai-based software techniques, december 2020.

[2] Edouard R. Batot. Second deliverable: Traceability language – definition and editor, december 2020.

[3] Edouard R. Batot. Tracea. `https://github.com/modelia/tracea`, 2021.

[4] Edouard R. Batot, Sebastien Gerard, and Jordi Cabot. A survey-driven feature model for software traceability. 2020. (under review) https://github.com/ebatot/TraceaDSL/blob/main/batot2020_SurveyDrivenFeatureModel.pdf.

[5] Edouard R. Batot, Sebastien Gerard, and Jordi Cabot. (Not) yet another metamodel for software traceability. 2021. (under review).

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

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

[8] J. Holtmann, J. P. Steghöfer, M. Rath, and D. Schmelter. Cutting through the jungle: Disambiguating model-based traceability terminology. In 2020 IEEE 28th International Requirements Engineering Conference (RE), pages 8–19, Aug 2020.

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

[10] Patrick Mäder and Jane Cleland-Huang. A visual traceability modeling language. In Dorina C. Petriu, Nicolas Rouquette, and Øystein Haugen, editors, Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I, volume 6394 of Lecture Notes in Computer Science, pages 226–240. Springer, 2010.

[11] Salome Maro, Anthony Anjorin, Rebekka Wohlrab, and Jan-Philipp Steghöfer. Traceability maintenance: Factors and guidelines. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, page 414–425. Association for Computing Machinery, 2016.

[12] Jan-Philipp Steghofer and Salome Maro. Tracea. `https://git.eclipse.org/c/capra/org.eclipse.capra.git/`, 2021.

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